NetworkX Reference

Release 3.0rc2.dev0

Aric Hagberg, Dan Schult, Pieter Swart

Jan 04, 2023

CONTENTS

CHAPTER

ONE

INTRODUCTION

The structure of NetworkX can be seen by the organization of its source code. The package provides classes for graph objects, generators to create standard graphs, IO routines for reading in existing datasets, algorithms to analyze the resulting networks and some basic drawing tools.

Most of the NetworkX API is provided by functions which take a graph object as an argument. Methods of the graph object are limited to basic manipulation and reporting. This provides modularity of code and documentation. It also makes it easier for newcomers to learn about the package in stages. The source code for each module is meant to be easy to read and reading this Python code is actually a good way to learn more about network algorithms, but we have put a lot of effort into making the documentation sufficient and friendly. If you have suggestions or questions please contact us by joining the NetworkX Google group.

Classes are named using `CamelCase` (capital letters at the start of each word). functions, methods and variable names are `lower_case_underscore` (lowercase with an underscore representing a space between words).

1.1 NetworkX Basics

After starting Python, import the networkx module with (the recommended way)

```
>>> import networkx as nx
```

To save repetition, in the documentation we assume that NetworkX has been imported this way.

If importing networkx fails, it means that Python cannot find the installed module. Check your installation and your `PYTHONPATH`.

The following basic graph types are provided as Python classes:

Graph
: This class implements an undirected graph. It ignores multiple edges between two nodes. It does allow self-loop edges between a node and itself.

DiGraph
: Directed graphs, that is, graphs with directed edges. Provides operations common to directed graphs, (a subclass of Graph).

MultiGraph
: A flexible graph class that allows multiple undirected edges between pairs of nodes. The additional flexibility leads to some degradation in performance, though usually not significant.

MultiDiGraph
: A directed version of a MultiGraph.

Empty graph-like objects are created with

```
>>> G = nx.Graph()
>>> G = nx.DiGraph()
>>> G = nx.MultiGraph()
>>> G = nx.MultiDiGraph()
```

All graph classes allow any hashable object as a node. Hashable objects include strings, tuples, integers, and more. Arbitrary edge attributes such as weights and labels can be associated with an edge.

The graph internal data structures are based on an adjacency list representation and implemented using Python *dictionary* datastructures. The graph adjacency structure is implemented as a Python dictionary of dictionaries; the outer dictionary is keyed by nodes to values that are themselves dictionaries keyed by neighboring node to the edge attributes associated with that edge. This "dict-of-dicts" structure allows fast addition, deletion, and lookup of nodes and neighbors in large graphs. The underlying datastructure is accessed directly by methods (the programming interface "API") in the class definitions. All functions, on the other hand, manipulate graph-like objects solely via those API methods and not by acting directly on the datastructure. This design allows for possible replacement of the 'dicts-of-dicts'-based datastructure with an alternative datastructure that implements the same methods.

1.2 Graphs

The first choice to be made when using NetworkX is what type of graph object to use. A graph (network) is a collection of nodes together with a collection of edges that are pairs of nodes. Attributes are often associated with nodes and/or edges. NetworkX graph objects come in different flavors depending on two main properties of the network:

- Directed: Are the edges **directed**? Does the order of the edge pairs (u, v) matter? A directed graph is specified by the "Di" prefix in the class name, e.g. `DiGraph()`. We make this distinction because many classical graph properties are defined differently for directed graphs.
- Multi-edges: Are multiple edges allowed between each pair of nodes? As you might imagine, multiple edges requires a different data structure, though clever users could design edge data attributes to support this functionality. We provide a standard data structure and interface for this type of graph using the prefix "Multi", e.g., `MultiGraph()`.

The basic graph classes are named: *Graph*, *DiGraph*, *MultiGraph*, and *MultiDiGraph*

1.2.1 Nodes and Edges

The next choice you have to make when specifying a graph is what kinds of nodes and edges to use.

If the topology of the network is all you care about then using integers or strings as the nodes makes sense and you need not worry about edge data. If you have a data structure already in place to describe nodes you can simply use that structure as your nodes provided it is hashable. If it is not hashable you can use a unique identifier to represent the node and assign the data as a *node attribute*.

Edges often have data associated with them. Arbitrary data can be associated with edges as an *edge attribute*. If the data is numeric and the intent is to represent a *weighted* graph then use the 'weight' keyword for the attribute. Some of the graph algorithms, such as Dijkstra's shortest path algorithm, use this attribute name by default to get the weight for each edge.

Attributes can be assigned to an edge by using keyword/value pairs when adding edges. You can use any keyword to name your attribute and can then query the edge data using that attribute keyword.

Once you've decided how to encode the nodes and edges, and whether you have an undirected/directed graph with or without multiedges you are ready to build your network.

1.3 Graph Creation

NetworkX graph objects can be created in one of three ways:

- Graph generators—standard algorithms to create network topologies.
- Importing data from pre-existing (usually file) sources.
- Adding edges and nodes explicitly.

Explicit addition and removal of nodes/edges is the easiest to describe. Each graph object supplies methods to manipulate the graph. For example,

```
>>> import networkx as nx
>>> G = nx.Graph()
>>> G.add_edge(1, 2)  # default edge data=1
>>> G.add_edge(2, 3, weight=0.9)  # specify edge data
```

Edge attributes can be anything:

```
>>> import math
>>> G.add_edge('y', 'x', function=math.cos)
>>> G.add_node(math.cos)  # any hashable can be a node
```

You can add many edges at one time:

```
>>> elist = [(1, 2), (2, 3), (1, 4), (4, 2)]
>>> G.add_edges_from(elist)
>>> elist = [('a', 'b', 5.0), ('b', 'c', 3.0), ('a', 'c', 1.0), ('c', 'd', 7.3)]
>>> G.add_weighted_edges_from(elist)
```

See the *Tutorial* for more examples.

Some basic graph operations such as union and intersection are described in the *operators module* documentation.

Graph generators such as `binomial_graph()` and `erdos_renyi_graph()` are provided in the *graph generators* subpackage.

For importing network data from formats such as GML, GraphML, edge list text files see the *reading and writing graphs* subpackage.

1.4 Graph Reporting

Class views provide basic reporting of nodes, neighbors, edges and degree. These views provide iteration over the properties as well as membership queries and data attribute lookup. The views refer to the graph data structure so changes to the graph are reflected in the views. This is analogous to dictionary views in Python 3. If you want to change the graph while iterating you will need to use e.g. `for e in list(G.edges):`. The views provide set-like operations, e.g. union and intersection, as well as dict-like lookup and iteration of the data attributes using `G.edges[u, v]['color']` and `for e, datadict in G.edges.items():`. Methods `G.edges.items()` and `G.edges.values()` are familiar from python dicts. In addition `G.edges.data()` provides specific attribute iteration e.g. `for e, e_color in G.edges.data('color'):`.

The basic graph relationship of an edge can be obtained in two ways. One can look for neighbors of a node or one can look for edges. We jokingly refer to people who focus on nodes/neighbors as node-centric and people who focus on edges as edge-centric. The designers of NetworkX tend to be node-centric and view edges as a relationship between nodes. You can see this by our choice of lookup notation like `G[u]` providing neighbors (adjacency) while edge lookup is `G.edges[u, v]`. Most data structures for sparse graphs are essentially adjacency lists and so fit this perspective. In the

end, of course, it doesn't really matter which way you examine the graph. `G.edges` removes duplicate representations of undirected edges while neighbor reporting across all nodes will naturally report both directions.

Any properties that are more complicated than edges, neighbors and degree are provided by functions. For example `nx.triangles(G, n)` gives the number of triangles which include node n as a vertex. These functions are grouped in the code and documentation under the term *algorithms*.

1.5 Algorithms

A number of graph algorithms are provided with NetworkX. These include shortest path, and breadth first search (see *traversal*), clustering and isomorphism algorithms and others. There are many that we have not developed yet too. If you implement a graph algorithm that might be useful for others please let us know through the NetworkX Google group or the Github Developer Zone.

As an example here is code to use Dijkstra's algorithm to find the shortest weighted path:

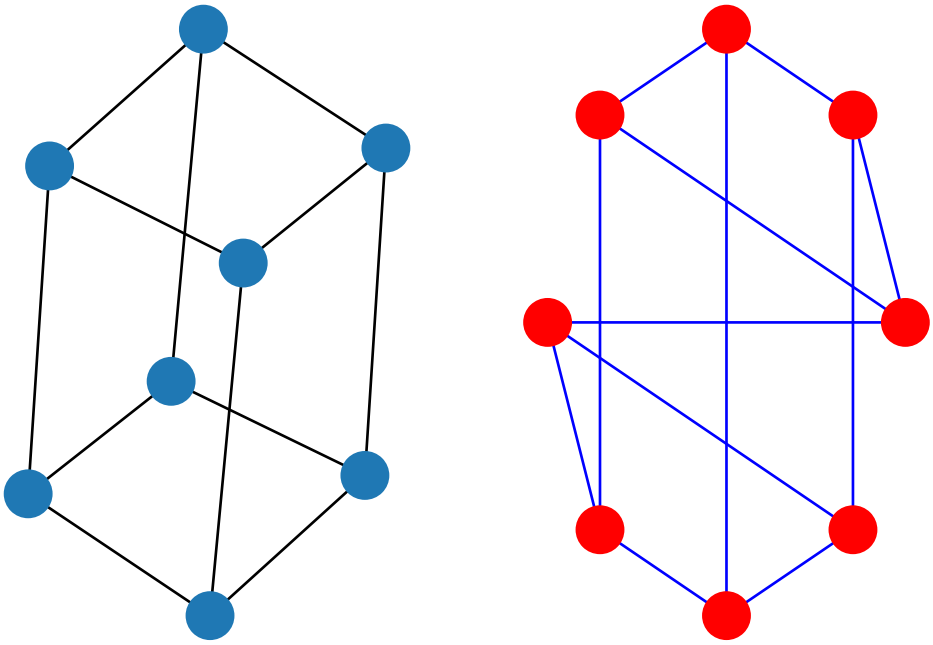
```
>>> G = nx.Graph()
>>> e = [('a', 'b', 0.3), ('b', 'c', 0.9), ('a', 'c', 0.5), ('c', 'd', 1.2)]
>>> G.add_weighted_edges_from(e)
>>> print(nx.dijkstra_path(G, 'a', 'd'))
['a', 'c', 'd']
```

1.6 Drawing

While NetworkX is not designed as a network drawing tool, we provide a simple interface to drawing packages and some simple layout algorithms. We interface to the excellent Graphviz layout tools like dot and neato with the (suggested) pygraphviz package or the pydot interface. Drawing can be done using external programs or the Matplotlib Python package. Interactive GUI interfaces are possible, though not provided. The drawing tools are provided in the module *drawing*.

The basic drawing functions essentially place the nodes on a scatterplot using the positions you provide via a dictionary or the positions are computed with a layout function. The edges are lines between those dots.

```
>>> import matplotlib.pyplot as plt
>>> G = nx.cubical_graph()
>>> subax1 = plt.subplot(121)
>>> nx.draw(G)   # default spring_layout
>>> subax2 = plt.subplot(122)
>>> nx.draw(G, pos=nx.circular_layout(G), node_color='r', edge_color='b')
```



See the examples for more ideas.

1.7 Data Structure

NetworkX uses a "dictionary of dictionaries of dictionaries" as the basic network data structure. This allows fast lookup with reasonable storage for large sparse networks. The keys are nodes so `G[u]` returns an adjacency dictionary keyed by neighbor to the edge attribute dictionary. A view of the adjacency data structure is provided by the dict-like object `G.adj` as e.g. `for node, nbrsdict in G.adj.items():`. The expression `G[u][v]` returns the edge attribute dictionary itself. A dictionary of lists would have also been possible, but not allow fast edge detection nor convenient storage of edge data.

Advantages of dict-of-dicts-of-dicts data structure:

- Find edges and remove edges with two dictionary look-ups.
- Prefer to "lists" because of fast lookup with sparse storage.
- Prefer to "sets" since data can be attached to edge.
- `G[u][v]` returns the edge attribute dictionary.
- `n in G` tests if node `n` is in graph `G`.
- `for n in G:` iterates through the graph.
- `for nbr in G[n]:` iterates through neighbors.

As an example, here is a representation of an undirected graph with the edges (A, B) and (B, C).

```
>>> G = nx.Graph()
>>> G.add_edge('A', 'B')
>>> G.add_edge('B', 'C')
>>> print(G.adj)
{'A': {'B': {}}, 'B': {'A': {}, 'C': {}}, 'C': {'B': {}}}
```

The data structure gets morphed slightly for each base graph class. For DiGraph two dict-of-dicts-of-dicts structures are provided, one for successors (`G.succ`) and one for predecessors (`G.pred`). For MultiGraph/MultiDiGraph we use a dict-of-dicts-of-dicts-of-dicts[1] where the third dictionary is keyed by an edge key identifier to the fourth dictionary which contains the edge attributes for that edge between the two nodes.

Graphs provide two interfaces to the edge data attributes: adjacency and edges. So `G[u][v]['width']` is the same as `G.edges[u, v]['width']`.

```
>>> G = nx.Graph()
>>> G.add_edge(1, 2, color='red', weight=0.84, size=300)
>>> print(G[1][2]['size'])
300
>>> print(G.edges[1, 2]['color'])
red
```

- ;
- ;
- .

[1] "It's dictionaries all the way down."

CHAPTER

TWO

GRAPH TYPES

NetworkX provides data structures and methods for storing graphs.

All NetworkX graph classes allow (hashable) Python objects as nodes and any Python object can be assigned as an edge attribute.

The choice of graph class depends on the structure of the graph you want to represent.

2.1 Which graph class should I use?

Networkx Class	Type	Self-loops allowed	Parallel edges allowed
Graph	undirected	Yes	No
DiGraph	directed	Yes	No
MultiGraph	undirected	Yes	Yes
MultiDiGraph	directed	Yes	Yes

2.2 Basic graph types

2.2.1 Graph—Undirected graphs with self loops

Overview

class Graph(*incoming_graph_data=None*, ***attr*)

Base class for undirected graphs.

A Graph stores nodes and edges with optional data, or attributes.

Graphs hold undirected edges. Self loops are allowed but multiple (parallel) edges are not.

Nodes can be arbitrary (hashable) Python objects with optional key/value attributes, except that `None` is not allowed as a node.

Edges are represented as links between nodes with optional key/value 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 any format that is supported by the to_networkx_graph() function, currently including edge list, dict of dicts, dict of lists, NetworkX graph, 2D NumPy array, SciPy sparse matrix, or PyGraphviz graph.

attr
: [keyword arguments, optional (default= no attributes)] Attributes to add to graph as key=value pairs.

See also:

DiGraph
MultiGraph
MultiDiGraph

Examples

Create an empty graph structure (a "null graph") with no nodes and no edges.

```
>>> G = nx.Graph()
```

G can be grown in several ways.

Nodes:

Add one node at a time:

```
>>> G.add_node(1)
```

Add the nodes from any container (a list, dict, set or even the lines from a file or the nodes from another graph).

```
>>> G.add_nodes_from([2, 3])
>>> G.add_nodes_from(range(100, 110))
>>> H = nx.path_graph(10)
>>> G.add_nodes_from(H)
```

In addition to strings and integers any hashable Python object (except None) can represent a node, e.g. a customized node object, or even another Graph.

```
>>> G.add_node(H)
```

Edges:

G can also be grown by adding edges.

Add one edge,

```
>>> G.add_edge(1, 2)
```

a list of edges,

```
>>> G.add_edges_from([(1, 2), (1, 3)])
```

or a collection of edges,

```
>>> G.add_edges_from(H.edges)
```

If some edges connect nodes not yet in the graph, the nodes are added automatically. There are no errors when adding nodes or edges that already exist.

Attributes:

Each graph, node, and edge can hold key/value attribute pairs in an associated attribute dictionary (the keys must be hashable). By default these are empty, but can be added or changed using add_edge, add_node or direct manipulation of the attribute dictionaries named graph, node and edge respectively.

```
>>> G = nx.Graph(day="Friday")
>>> G.graph
{'day': 'Friday'}
```

Add node attributes using add_node(), add_nodes_from() or G.nodes

```
>>> G.add_node(1, time="5pm")
>>> G.add_nodes_from([3], time="2pm")
>>> G.nodes[1]
{'time': '5pm'}
>>> G.nodes[1]["room"] = 714  # node must exist already to use G.nodes
>>> del G.nodes[1]["room"]  # remove attribute
>>> list(G.nodes(data=True))
[(1, {'time': '5pm'}), (3, {'time': '2pm'})]
```

Add edge attributes using add_edge(), add_edges_from(), subscript notation, or G.edges.

```
>>> G.add_edge(1, 2, weight=4.7)
>>> G.add_edges_from([(3, 4), (4, 5)], color="red")
>>> G.add_edges_from([(1, 2, {"color": "blue"}), (2, 3, {"weight": 8})])
>>> G[1][2]["weight"] = 4.7
>>> G.edges[1, 2]["weight"] = 4
```

Warning: we protect the graph data structure by making `G.edges` a read-only dict-like structure. However, you can assign to attributes in e.g. `G.edges[1, 2]`. Thus, use 2 sets of brackets to add/change data attributes: `G.edges[1, 2]['weight'] = 4` (For multigraphs: `MG.edges[u, v, key][name] = value`).

Shortcuts:

Many common graph features allow python syntax to speed reporting.

```
>>> 1 in G  # check if node in graph
True
>>> [n for n in G if n < 3]  # iterate through nodes
[1, 2]
>>> len(G)  # number of nodes in graph
5
```

Often the best way to traverse all edges of a graph is via the neighbors. The neighbors are reported as an adjacency-dict `G.adj` or `G.adjacency()`

```
>>> for n, nbrsdict in G.adjacency():
...     for nbr, eattr in nbrsdict.items():
...         if "weight" in eattr:
...             # Do something useful with the edges
...             pass
```

But the edges() method is often more convenient:

```
>>> for u, v, weight in G.edges.data("weight"):
...     if weight is not None:
...         # Do something useful with the edges
...         pass
```

Reporting:

Simple graph information is obtained using object-attributes and methods. Reporting typically provides views instead of containers to reduce memory usage. The views update as the graph is updated similarly to dict-views. The objects *nodes*, *edges* and *adj* provide access to data attributes via lookup (e.g. `nodes[n]`, `edges[u, v]`, `adj[u][v]`) and iteration (e.g. `nodes.items()`, `nodes.data('color')`, `nodes.data('color', default='blue')` and similarly for *edges*) Views exist for *nodes*, *edges*, *neighbors()*/*adj* and *degree*.

For details on these and other miscellaneous methods, see below.

Subclasses (Advanced):

The Graph class uses a dict-of-dict-of-dict data structure. The outer dict (node_dict) holds adjacency information keyed by node. The next dict (adjlist_dict) represents the adjacency information and holds 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
```

Methods

Adding and removing nodes and edges

Graph.__init__([incoming_graph_data])	Initialize a graph with edges, name, or graph attributes.
Graph.add_node(node_for_adding, **attr)	Add a single node `node_for_adding` and update node attributes.
Graph.add_nodes_from(nodes_for_adding, **attr)	Add multiple nodes.
Graph.remove_node(n)	Remove node n.
Graph.remove_nodes_from(nodes)	Remove multiple nodes.
Graph.add_edge(u_of_edge, v_of_edge, **attr)	Add an edge between u and v.
Graph.add_edges_from(ebunch_to_add, **attr)	Add all the edges in ebunch_to_add.
Graph.add_weighted_edges_from(ebunch_to_add)	Add weighted edges in `ebunch_to_add` with specified weight attr
Graph.remove_edge(u, v)	Remove the edge between u and v.
Graph.remove_edges_from(ebunch)	Remove all edges specified in ebunch.
Graph.update([edges, nodes])	Update the graph using nodes/edges/graphs as input.
Graph.clear()	Remove all nodes and edges from the graph.
Graph.clear_edges()	Remove all edges from the graph without altering nodes.

Graph.__init__

Graph.**__init__**(*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'}
```

Graph.add_node

Graph.**add_node**(*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

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.

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))
```

Graph.add_nodes_from

Graph.**add_nodes_from**(*nodes_for_adding*, ***attr*)

Add multiple nodes.

Parameters

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))
```

Graph.remove_node

Graph.**remove_node**(*n*)

Remove node n.

Removes the node n and all adjacent edges. Attempting to remove a non-existent node will raise an exception.

Parameters

n
[node] A node in the graph

Raises

NetworkXError
If n is not in the graph.

See also:

remove_nodes_from

Examples

```
>>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> list(G.edges)
[(0, 1), (1, 2)]
>>> G.remove_node(1)
>>> list(G.edges)
[]
```

Graph.remove_nodes_from

Graph.**remove_nodes_from**(*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`.

Examples

```
>>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> e = list(G.nodes)
>>> e
[0, 1, 2]
>>> G.remove_nodes_from(e)
>>> list(G.nodes)
[]
```

Evaluate an iterator over a graph if using it to modify the same graph

```
>>> G = nx.Graph([(0, 1), (1, 2), (3, 4)])
>>> # this command will fail, as the graph's dict is modified during iteration
>>> # G.remove_nodes_from(n for n in G.nodes if n < 2)
>>> # this command will work, since the dictionary underlying graph is not␣
↪modified
>>> G.remove_nodes_from(list(n for n in G.nodes if n < 2))
```

Graph.add_edge

`Graph.`**`add_edge`**(*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})
```

Graph.add_edges_from

`Graph.`**`add_edges_from`**(*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.Graph([(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))
>>> # correct way - note that there will be no self-edge for node 5
>>> G.add_edges_from(list((5, n) for n in G.nodes))
```

Graph.add_weighted_edges_from

`Graph.`**`add_weighted_edges_from`**(*ebunch_to_add*, *weight='weight'*, ***attr*)

Add weighted edges in `ebunch_to_add` with specified weight attr

Parameters

ebunch_to_add
: [container of edges] Each edge given in the list or container will be added to the graph. The edges must be given as 3-tuples (u, v, w) where w is a number.

weight
: [string, optional (default= 'weight')] The attribute name for the edge weights to be added.

attr
: [keyword arguments, optional (default= no attributes)] Edge attributes to add/update for all edges.

See also:

add_edge
: add a single edge

add_edges_from
: add multiple edges

Notes

Adding the same edge twice for Graph/DiGraph simply updates the edge data. For MultiGraph/MultiDiGraph, duplicate edges are stored.

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_weighted_edges_from`.

Examples

```
>>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_weighted_edges_from([(0, 1, 3.0), (1, 2, 7.5)])
```

Evaluate an iterator over edges before passing it

```
>>> G = nx.Graph([(1, 2), (2, 3), (3, 4)])
>>> weight = 0.1
>>> # Grow graph by one new node, adding edges to all existing nodes.
>>> # wrong way - will raise RuntimeError
>>> # G.add_weighted_edges_from(((5, n, weight) for n in G.nodes))
>>> # correct way - note that there will be no self-edge for node 5
>>> G.add_weighted_edges_from(list((5, n, weight) for n in G.nodes))
```

Graph.remove_edge

Graph.**remove_edge**(*u*, *v*)

Remove the edge between u and v.

Parameters

u, v
[nodes] Remove the edge between nodes u and v.

Raises

NetworkXError
If there is not an edge between u and v.

See also:

remove_edges_from
remove a collection of edges

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, etc
>>> G.remove_edge(0, 1)
>>> e = (1, 2)
>>> G.remove_edge(*e)  # unpacks e from an edge tuple
>>> e = (2, 3, {"weight": 7})  # an edge with attribute data
>>> G.remove_edge(*e[:2])  # select first part of edge tuple
```

Graph.remove_edges_from

Graph.**remove_edges_from**(*ebunch*)

Remove all edges specified in ebunch.

Parameters

ebunch: list or container of edge tuples
Each edge given in the list or container will be removed from the graph. The edges can be:

- 2-tuples (u, v) edge between u and v.
- 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)
```

Graph.update

Graph.**update**(*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.

See also:

add_edges_from
: add multiple edges to a graph

add_nodes_from
: add multiple nodes to a graph

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}}
>>> e = [
...     (u, v, {"weight": d})
...     for u, nbrs in adj.items()
...     for v, d in nbrs.items()
... ]
>>> DG.update(edges=e, nodes=adj)
```

```
>>> # dict-of-dict-of-dict
>>> adj = {1: {2: {"weight": 1.3}, 3: {"color": 0.7, "weight": 1.2}}}
>>> e = [
...     (u, v, {"weight": d})
...     for u, nbrs in adj.items()
...     for v, d in nbrs.items()
... ]
>>> DG.update(edges=e, nodes=adj)
```

```
>>> # predecessor adjacency (dict-of-set)
>>> pred = {1: {2, 3}, 2: {3}, 3: {3}}
>>> e = [(v, u) for u, nbrs in pred.items() for v in nbrs]
```

```
>>> # MultiGraph dict-of-dict-of-dict-of-attribute
>>> MDG = nx.MultiDiGraph()
>>> adj = {
...     1: {2: {0: {"weight": 1.3}, 1: {"weight": 1.2}}},
...     3: {2: {0: {"weight": 0.7}}},
... }
>>> e = [
...     (u, v, ekey, d)
...     for u, nbrs in adj.items()
...     for v, keydict in nbrs.items()
...     for ekey, d in keydict.items()
... ]
>>> MDG.update(edges=e)
```

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)
```

Graph.clear

Graph.**clear**()

Remove all nodes and edges from the graph.

This also removes the name, and all graph, node, and edge attributes.

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.clear()
>>> list(G.nodes)
[]
>>> list(G.edges)
[]
```

Graph.clear_edges

Graph.**clear_edges**()

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)
[]
```

Reporting nodes edges and neighbors

Graph.nodes	A NodeView of the Graph as G.nodes or G.nodes().
Graph.__iter__()	Iterate over the nodes.
Graph.has_node(n)	Returns True if the graph contains the node n.
Graph.__contains__(n)	Returns True if n is a node, False otherwise.
Graph.edges	An EdgeView of the Graph as G.edges or G.edges().
Graph.has_edge(u, v)	Returns True if the edge (u, v) is in the graph.
Graph.get_edge_data(u, v[, default])	Returns the attribute dictionary associated with edge (u, v).
Graph.neighbors(n)	Returns an iterator over all neighbors of node n.
Graph.adj	Graph adjacency object holding the neighbors of each node.
Graph.__getitem__(n)	Returns a dict of neighbors of node n.
Graph.adjacency()	Returns an iterator over (node, adjacency dict) tuples for all nodes.
Graph.nbunch_iter([nbunch])	Returns an iterator over nodes contained in nbunch that are also in the graph.

Graph.nodes

property Graph.**nodes**

A NodeView of the Graph as G.nodes or G.nodes().

Can be used as `G.nodes` for data lookup and for set-like operations. Can also be used as `G.nodes(data='color', default=None)` to return a NodeDataView which reports specific node data but no set operations. It presents a dict-like interface as well with `G.nodes.items()` iterating over `(node, nodedata)` 2-tuples and `G.nodes[3]['foo']` providing the value of the `foo` attribute for node `3`. In addition, a view `G.nodes.data('foo')` provides a dict-like interface to the `foo` attribute of each node. `G.nodes.data('foo', default=1)` provides a default for nodes that do not have attribute `foo`.

Parameters

data

[string or bool, optional (default=False)] The node attribute returned in 2-tuple (n, ddict[data]). If True, return entire node attribute dict as (n, ddict). If False, return just the nodes n.

default
[value, optional (default=None)] Value used for nodes that don't have the requested attribute. Only relevant if data is not True or False.

Returns

NodeView
Allows set-like operations over the nodes as well as node attribute dict lookup and calling to get a NodeDataView. A NodeDataView iterates over `(n, data)` and has no set operations. A NodeView iterates over `n` and includes set operations.

When called, if data is False, an iterator over nodes. Otherwise an iterator of 2-tuples (node, attribute value) where the attribute is specified in `data`. If data is True then the attribute becomes the entire data dictionary.

Notes

If your node data is not needed, it is simpler and equivalent to use the expression `for n in G`, or `list(G)`.

Examples

There are two simple ways of getting a list of all nodes in the graph:

```
>>> G = nx.path_graph(3)
>>> list(G.nodes)
[0, 1, 2]
>>> list(G)
[0, 1, 2]
```

To get the node data along with the nodes:

```
>>> G.add_node(1, time="5pm")
>>> G.nodes[0]["foo"] = "bar"
>>> list(G.nodes(data=True))
[(0, {'foo': 'bar'}), (1, {'time': '5pm'}), (2, {})]
>>> list(G.nodes.data())
[(0, {'foo': 'bar'}), (1, {'time': '5pm'}), (2, {})]
```

```
>>> list(G.nodes(data="foo"))
[(0, 'bar'), (1, None), (2, None)]
>>> list(G.nodes.data("foo"))
[(0, 'bar'), (1, None), (2, None)]
```

```
>>> list(G.nodes(data="time"))
[(0, None), (1, '5pm'), (2, None)]
>>> list(G.nodes.data("time"))
[(0, None), (1, '5pm'), (2, None)]
```

```
>>> list(G.nodes(data="time", default="Not Available"))
[(0, 'Not Available'), (1, '5pm'), (2, 'Not Available')]
>>> list(G.nodes.data("time", default="Not Available"))
[(0, 'Not Available'), (1, '5pm'), (2, 'Not Available')]
```

If some of your nodes have an attribute and the rest are assumed to have a default attribute value you can create a dictionary from node/attribute pairs using the `default` keyword argument to guarantee the value is never None:

```
>>> G = nx.Graph()
>>> G.add_node(0)
>>> G.add_node(1, weight=2)
>>> G.add_node(2, weight=3)
>>> dict(G.nodes(data="weight", default=1))
{0: 1, 1: 2, 2: 3}
```

Graph.__iter__

Graph.**__iter__**()

Iterate over the nodes. Use: 'for n in G'.

Returns

niter
[iterator] An iterator over all nodes in the graph.

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> [n for n in G]
[0, 1, 2, 3]
>>> list(G)
[0, 1, 2, 3]
```

Graph.has_node

Graph.**has_node**(*n*)

Returns True if the graph contains the node n.

Identical to `n in G`

Parameters

n
[node]

Examples

```
>>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.has_node(0)
True
```

It is more readable and simpler to use

```
>>> 0 in G
True
```

Graph.__contains__

Graph.**__contains__**(*n*)

Returns True if n is a node, False otherwise. Use: ‘n in G’.

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> 1 in G
True
```

Graph.edges

property Graph.**edges**

An EdgeView of the Graph as G.edges or G.edges().

edges(self, nbunch=None, data=False, default=None)

The EdgeView provides set-like operations on the edge-tuples as well as edge attribute lookup. When called, it also provides an EdgeDataView object which allows control of access to edge attributes (but does not provide set-like operations). Hence, `G.edges[u, v]['color']` provides the value of the color attribute for edge `(u, v)` while `for (u, v, c) in G.edges.data('color', default='red'):` iterates through all the edges yielding the color attribute with default `'red'` if no color attribute exists.

Parameters

nbunch
: [single node, container, or all nodes (default= all nodes)] The view will only report edges from 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).

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

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)])
```

Graph.has_edge

Graph.**has_edge**(*u*, *v*)

Returns True if the edge (u, v) is in the graph.

This is the same as `v in G[u]` without KeyError exceptions.

Parameters

u, v
: [nodes] Nodes can be, for example, strings or numbers. Nodes must be hashable (and not None) Python objects.

Returns

edge_ind
: [bool] True if edge is in the graph, False otherwise.

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.has_edge(0, 1)  # using two nodes
True
>>> e = (0, 1)
>>> G.has_edge(*e)  #  e is a 2-tuple (u, v)
True
>>> e = (0, 1, {"weight": 7})
>>> G.has_edge(*e[:2])  # e is a 3-tuple (u, v, data_dictionary)
True
```

The following syntax are equivalent:

```
>>> G.has_edge(0, 1)
True
>>> 1 in G[0]  # though this gives KeyError if 0 not in G
True
```

Graph.get_edge_data

Graph.**get_edge_data**(*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
```

Graph.neighbors

Graph.**neighbors**(*n*)

Returns an iterator over all neighbors of node n.

This is identical to `iter(G[n])`

Parameters

n
[node] A node in the graph

Returns

neighbors
[iterator] An iterator over all neighbors of node n

Raises

NetworkXError
If the node n is not in the graph.

Notes

Alternate ways to access the neighbors are `G.adj[n]` or `G[n]`:

```
>>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge("a", "b", weight=7)
>>> G["a"]
AtlasView({'b': {'weight': 7}})
>>> G = nx.path_graph(4)
>>> [n for n in G[0]]
[1]
```

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> [n for n in G.neighbors(0)]
[1]
```

Graph.adj

property Graph.**adj**

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.

Graph.__getitem__

Graph.**__getitem__**(*n*)

Returns a dict of neighbors of node n. Use: 'G[n]'.

Parameters

n
[node] A node in the graph.

Returns

adj_dict
[dictionary] The adjacency dictionary for nodes connected to n.

Notes

G[n] is the same as G.adj[n] and similar to G.neighbors(n) (which is an iterator over G.adj[n])

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G[0]
AtlasView({1: {}})
```

Graph.adjacency

`Graph.`**`adjacency`**`()`

Returns an iterator over (node, adjacency dict) tuples for all nodes.

For directed graphs, only outgoing neighbors/adjacencies are included.

Returns

adj_iter
[iterator] An iterator over (node, adjacency dictionary) for all nodes in the graph.

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> [(n, nbrdict) for n, nbrdict in G.adjacency()]
[(0, {1: {}}), (1, {0: {}, 2: {}}), (2, {1: {}, 3: {}}), (3, {2: {}})]
```

Graph.nbunch_iter

`Graph.`**`nbunch_iter`**`(`*nbunch=None*`)`

Returns an iterator over nodes contained in nbunch that are also in the graph.

The nodes in nbunch are checked for membership in the graph and if not are silently ignored.

Parameters

nbunch
[single node, container, or all nodes (default= all nodes)] The view will only report edges incident to these nodes.

Returns

niter
[iterator] An iterator over nodes in nbunch that are also in the graph. If nbunch is None, iterate over all nodes in the graph.

Raises

NetworkXError
If nbunch is not a node or sequence of nodes. If a node in nbunch is not hashable.

See also:

Graph.__iter__

Notes

When nbunch is an iterator, the returned iterator yields values directly from nbunch, becoming exhausted when nbunch is exhausted.

To test whether nbunch is a single node, one can use "if nbunch in self:", even after processing with this routine.

If nbunch is not a node or a (possibly empty) sequence/iterator or None, a *NetworkXError* is raised. Also, if any object in nbunch is not hashable, a *NetworkXError* is raised.

Counting nodes edges and neighbors

Graph.order()	Returns the number of nodes in the graph.
Graph.number_of_nodes()	Returns the number of nodes in the graph.
Graph.__len__()	Returns the number of nodes in the graph.
Graph.degree	A DegreeView for the Graph as G.degree or G.degree().
Graph.size([weight])	Returns the number of edges or total of all edge weights.
Graph.number_of_edges([u, v])	Returns the number of edges between two nodes.

Graph.order

Graph.**order**()

Returns the number of nodes in the graph.

Returns

nnodes
[int] The number of nodes in the graph.

See also:

number_of_nodes
identical method

__len__
identical method

Examples

```
>>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.order()
3
```

Graph.number_of_nodes

Graph.**number_of_nodes**()

Returns the number of nodes in the graph.

Returns

nnodes
[int] The number of nodes in the graph.

See also:

order
identical method

__len__
identical method

Examples

```
>>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.number_of_nodes()
3
```

Graph.__len__

Graph.**__len__**()

Returns the number of nodes in the graph. Use: 'len(G)'.

Returns

nnodes
[int] The number of nodes in the graph.

See also:

number_of_nodes
identical method

order
identical method

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> len(G)
4
```

Graph.degree

property Graph.**degree**

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

DegreeView or int
If multiple nodes are requested (the default), returns a `DegreeView` mapping nodes to their degree. If a single node is requested, returns the degree of the node as an integer.

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.degree[0]  # node 0 has degree 1
1
>>> list(G.degree([0, 1, 2]))
[(0, 1), (1, 2), (2, 2)]
```

Graph.size

Graph.**size**(*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
```

Graph.number_of_edges

Graph.**number_of_edges**(*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
```

Making copies and subgraphs

Graph.copy([as_view])	Returns a copy of the graph.
Graph.to_undirected([as_view])	Returns an undirected copy of the graph.
Graph.to_directed([as_view])	Returns a directed representation of the graph.
Graph.subgraph(nodes)	Returns a SubGraph view of the subgraph induced on `nodes`.
Graph.edge_subgraph(edges)	Returns the subgraph induced by the specified edges.

Graph.copy

`Graph.`**`copy`**(*as_view=False*)

Returns a copy of the graph.

The copy method by default returns an independent shallow copy of the graph and attributes. That is, if an attribute is a container, that container is shared by the original an the copy. Use Python's `copy.deepcopy` for new containers.

If `as_view` is True then a view is returned instead of a copy.

Parameters

as_view
: [bool, optional (default=False)] If True, the returned graph-view provides a read-only view of the original graph without actually copying any data.

Returns

G
: [Graph] A copy of the graph.

See also:

to_directed
return a directed copy of the graph.

Notes

All copies reproduce the graph structure, but data attributes may be handled in different ways. There are four types of copies of a graph that people might want.

Deepcopy – A "deepcopy" copies the graph structure as well as all data attributes and any objects they might contain. The entire graph object is new so that changes in the copy do not affect the original object. (see Python's copy.deepcopy)

Data Reference (Shallow) – For a shallow copy the graph structure is copied but the edge, node and graph attribute dicts are references to those in the original graph. This saves time and memory but could cause confusion if you change an attribute in one graph and it changes the attribute in the other. NetworkX does not provide this level of shallow copy.

Independent Shallow – This copy creates new independent attribute dicts and then does a shallow copy of the attributes. That is, any attributes that are containers are shared between the new graph and the original. This is exactly what `dict.copy()` provides. You can obtain this style copy using:

```
>>> G = nx.path_graph(5)
>>> H = G.copy()
>>> H = G.copy(as_view=False)
>>> H = nx.Graph(G)
>>> H = G.__class__(G)
```

Fresh Data – For fresh data, the graph structure is copied while new empty data attribute dicts are created. The resulting graph is independent of the original and it has no edge, node or graph attributes. Fresh copies are not enabled. Instead use:

```
>>> H = G.__class__()
>>> H.add_nodes_from(G)
>>> H.add_edges_from(G.edges)
```

View – Inspired by dict-views, graph-views act like read-only versions of the original graph, providing a copy of the original structure without requiring any memory for copying the information.

See the Python copy module for more information on shallow and deep copies, https://docs.python.org/3/library/copy.html.

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> H = G.copy()
```

Graph.to_undirected

Graph.**to_undirected**(*as_view=False*)

Returns an undirected copy of the graph.

Parameters

as_view
[bool (optional, default=False)] If True return a view of the original undirected graph.

Returns

G
[Graph/MultiGraph] A deepcopy of the graph.

See also:

Graph, ***copy***, ***add_edge***, ***add_edges_from***

Notes

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 = nx.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)]
```

Graph.to_directed

Graph.**to_directed**(*as_view=False*)

Returns a directed representation of the graph.

Returns

G
[DiGraph] A directed graph with the same name, same nodes, and with each edge (u, v, data) replaced by two directed edges (u, v, data) and (v, u, data).

Notes

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 D=DiGraph(G) 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 Graph to use dict-like objects in the data structure, those changes do not transfer to the DiGraph created by this method.

Examples

```
>>> G = nx.Graph()  # or MultiGraph, etc
>>> G.add_edge(0, 1)
>>> H = G.to_directed()
>>> list(H.edges)
[(0, 1), (1, 0)]
```

If already directed, return a (deep) copy

```
>>> G = nx.DiGraph()  # or MultiDiGraph, etc
>>> G.add_edge(0, 1)
>>> H = G.to_directed()
>>> list(H.edges)
[(0, 1)]
```

Graph.subgraph

Graph.**subgraph**(*nodes*)

Returns a SubGraph view of the subgraph induced on *nodes*.

The induced subgraph of the graph contains the nodes in *nodes* and the edges between those nodes.

Parameters

nodes
[list, iterable] A container of nodes which will be iterated through once.

Returns

G
[SubGraph View] A subgraph view of the graph. The graph structure cannot be changed but node/edge attributes can and are shared with the original graph.

Notes

The graph, edge and node attributes are shared with the original graph. Changes to the graph structure is ruled out by the view, but changes to attributes are reflected in the original graph.

To create a subgraph with its own copy of the edge/node attributes use: G.subgraph(nodes).copy()

For an inplace reduction of a graph to a subgraph you can remove nodes: G.remove_nodes_from([n for n in G if n not in set(nodes)])

Subgraph views are sometimes NOT what you want. In most cases where you want to do more than simply look at the induced edges, it makes more sense to just create the subgraph as its own graph with code like:

```
# Create a subgraph SG based on a (possibly multigraph) G
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)]
```

Graph.edge_subgraph

Graph.**edge_subgraph**(*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)]
```

2.2.2 DiGraph—Directed graphs with self loops

Overview

class DiGraph(*incoming_graph_data=None*, ***attr*)

Base class for directed graphs.

A DiGraph stores nodes and edges with optional data, or attributes.

DiGraphs hold directed edges. Self loops are allowed but multiple (parallel) edges are not.

Nodes can be arbitrary (hashable) Python objects with optional key/value attributes. By convention `None` is not used as a node.

Edges are represented as links between nodes with optional key/value 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 any format that is supported by the to_networkx_graph() function, currently including edge list, dict of dicts, dict of lists, NetworkX graph, 2D NumPy array, SciPy sparse matrix, or PyGraphviz graph.

attr
: [keyword arguments, optional (default= no attributes)] Attributes to add to graph as key=value pairs.

See also:

Graph
MultiGraph
MultiDiGraph

Examples

Create an empty graph structure (a "null graph") with no nodes and no edges.

```
>>> G = nx.DiGraph()
```

G can be grown in several ways.

Nodes:

Add one node at a time:

```
>>> G.add_node(1)
```

Add the nodes from any container (a list, dict, set or even the lines from a file or the nodes from another graph).

```
>>> G.add_nodes_from([2, 3])
>>> G.add_nodes_from(range(100, 110))
>>> H = nx.path_graph(10)
>>> G.add_nodes_from(H)
```

In addition to strings and integers any hashable Python object (except None) can represent a node, e.g. a customized node object, or even another Graph.

```
>>> G.add_node(H)
```

Edges:

G can also be grown by adding edges.

Add one edge,

```
>>> G.add_edge(1, 2)
```

a list of edges,

```
>>> G.add_edges_from([(1, 2), (1, 3)])
```

or a collection of edges,

```
>>> G.add_edges_from(H.edges)
```

If some edges connect nodes not yet in the graph, the nodes are added automatically. There are no errors when adding nodes or edges that already exist.

Attributes:

Each graph, node, and edge can hold key/value attribute pairs in an associated attribute dictionary (the keys must be hashable). By default these are empty, but can be added or changed using add_edge, add_node or direct manipulation of the attribute dictionaries named graph, node and edge respectively.

```
>>> G = nx.DiGraph(day="Friday")
>>> G.graph
{'day': 'Friday'}
```

Add node attributes using add_node(), add_nodes_from() or G.nodes

```
>>> G.add_node(1, time="5pm")
>>> G.add_nodes_from([3], time="2pm")
>>> G.nodes[1]
{'time': '5pm'}
>>> G.nodes[1]["room"] = 714
>>> del G.nodes[1]["room"]  # remove attribute
>>> list(G.nodes(data=True))
[(1, {'time': '5pm'}), (3, {'time': '2pm'})]
```

Add edge attributes using add_edge(), add_edges_from(), subscript notation, or G.edges.

```
>>> G.add_edge(1, 2, weight=4.7)
>>> G.add_edges_from([(3, 4), (4, 5)], color="red")
>>> G.add_edges_from([(1, 2, {"color": "blue"}), (2, 3, {"weight": 8})])
>>> G[1][2]["weight"] = 4.7
>>> G.edges[1, 2]["weight"] = 4
```

Warning: we protect the graph data structure by making `G.edges[1, 2]` a read-only dict-like structure. However, you can assign to attributes in e.g. `G.edges[1, 2]`. Thus, use 2 sets of brackets to add/change data attributes: `G.edges[1, 2]['weight'] = 4` (For multigraphs: `MG.edges[u, v, key][name] = value`).

Shortcuts:

Many common graph features allow python syntax to speed reporting.

```
>>> 1 in G  # check if node in graph
True
>>> [n for n in G if n < 3]  # iterate through nodes
[1, 2]
>>> len(G)  # number of nodes in graph
5
```

Often the best way to traverse all edges of a graph is via the neighbors. The neighbors are reported as an adjacency-dict `G.adj` or `G.adjacency()`

```
>>> for n, nbrsdict in G.adjacency():
...     for nbr, eattr in nbrsdict.items():
...         if "weight" in eattr:
...             # Do something useful with the edges
...             pass
```

But the edges reporting object is often more convenient:

```
>>> for u, v, weight in G.edges(data="weight"):
...     if weight is not None:
...         # Do something useful with the edges
...         pass
```

Reporting:

Simple graph information is obtained using object-attributes and methods. Reporting usually provides views instead of containers to reduce memory usage. The views update as the graph is updated similarly to dict-views. The objects *nodes*, *edges* and *adj* provide access to data attributes via lookup (e.g. `nodes[n]`, `edges[u, v]`, `adj[u][v]`) and iteration (e.g. `nodes.items()`, `nodes.data('color')`, `nodes.data('color', default='blue')` and similarly for *edges*) Views exist for *nodes*, *edges*, *neighbors()*/*adj* and *degree*.

For details on these and other miscellaneous methods, see below.

Subclasses (Advanced):

The Graph class uses a dict-of-dict-of-dict data structure. The outer dict (node_dict) holds adjacency information keyed by node. The next dict (adjlist_dict) represents the adjacency information and holds 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. The variable names are node_dict_factory, node_attr_dict_factory, adjlist_inner_dict_factory, adjlist_outer_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, optional (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, optional (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
```

(continues on next page)

(continued from previous page)

```
...
...     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
```

Methods

Adding and removing nodes and edges

DiGraph.__init__([incoming_graph_data])	Initialize a graph with edges, name, or graph attributes.
DiGraph.add_node(node_for_adding, **attr)	Add a single node `node_for_adding` and update node attributes.
DiGraph.add_nodes_from(nodes_for_adding, **attr)	Add multiple nodes.
DiGraph.remove_node(n)	Remove node n.
DiGraph.remove_nodes_from(nodes)	Remove multiple nodes.
DiGraph.add_edge(u_of_edge, v_of_edge, **attr)	Add an edge between u and v.
DiGraph.add_edges_from(ebunch_to_add, **attr)	Add all the edges in ebunch_to_add.
DiGraph.add_weighted_edges_from(ebunch_to_add)	Add weighted edges in `ebunch_to_add` with specified weight attr
DiGraph.remove_edge(u, v)	Remove the edge between u and v.
DiGraph.remove_edges_from(ebunch)	Remove all edges specified in ebunch.
DiGraph.update([edges, nodes])	Update the graph using nodes/edges/graphs as input.
DiGraph.clear()	Remove all nodes and edges from the graph.
DiGraph.clear_edges()	Remove all edges from the graph without altering nodes.

DiGraph.__init__

DiGraph.**__init__**(*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'}
```

DiGraph.add_node

DiGraph.**add_node**(*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

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.

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))
```

DiGraph.add_nodes_from

DiGraph.**add_nodes_from**(*nodes_for_adding*, ***attr*)

Add multiple nodes.

Parameters

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.DiGraph([(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))
```

DiGraph.remove_node

DiGraph.**remove_node**(*n*)

Remove node n.

Removes the node n and all adjacent edges. Attempting to remove a non-existent node will raise an exception.

Parameters

n
[node] A node in the graph

Raises

NetworkXError
If n is not in the graph.

See also:

remove_nodes_from

Examples

```
>>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> list(G.edges)
[(0, 1), (1, 2)]
>>> G.remove_node(1)
>>> list(G.edges)
[]
```

DiGraph.remove_nodes_from

DiGraph.**remove_nodes_from**(*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`.

Examples

```
>>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> e = list(G.nodes)
>>> e
[0, 1, 2]
>>> G.remove_nodes_from(e)
>>> list(G.nodes)
[]
```

Evaluate an iterator over a graph if using it to modify the same graph

```
>>> G = nx.DiGraph([(0, 1), (1, 2), (3, 4)])
>>> # this command will fail, as the graph's dict is modified during iteration
>>> # G.remove_nodes_from(n for n in G.nodes if n < 2)
>>> # this command will work, since the dictionary underlying graph is not␣
↪modified
>>> G.remove_nodes_from(list(n for n in G.nodes if n < 2))
```

DiGraph.add_edge

`DiGraph.`**`add_edge`**(*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})
```

DiGraph.add_edges_from

DiGraph.**add_edges_from**(*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))
```

DiGraph.add_weighted_edges_from

`DiGraph.`**`add_weighted_edges_from`**(*ebunch_to_add*, *weight='weight'*, ***attr*)

Add weighted edges in `ebunch_to_add` with specified weight attr

Parameters

ebunch_to_add
: [container of edges] Each edge given in the list or container will be added to the graph. The edges must be given as 3-tuples (u, v, w) where w is a number.

weight
: [string, optional (default= 'weight')] The attribute name for the edge weights to be added.

attr
: [keyword arguments, optional (default= no attributes)] Edge attributes to add/update for all edges.

See also:

add_edge
: add a single edge

add_edges_from
: add multiple edges

Notes

Adding the same edge twice for Graph/DiGraph simply updates the edge data. For MultiGraph/MultiDiGraph, duplicate edges are stored.

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_weighted_edges_from`.

Examples

```
>>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_weighted_edges_from([(0, 1, 3.0), (1, 2, 7.5)])
```

Evaluate an iterator over edges before passing it

```
>>> G = nx.Graph([(1, 2), (2, 3), (3, 4)])
>>> weight = 0.1
>>> # Grow graph by one new node, adding edges to all existing nodes.
>>> # wrong way - will raise RuntimeError
>>> # G.add_weighted_edges_from(((5, n, weight) for n in G.nodes))
>>> # correct way - note that there will be no self-edge for node 5
>>> G.add_weighted_edges_from(list((5, n, weight) for n in G.nodes))
```

DiGraph.remove_edge

DiGraph.**remove_edge**(*u*, *v*)

Remove the edge between u and v.

Parameters

u, v
[nodes] Remove the edge between nodes u and v.

Raises

NetworkXError
If there is not an edge between u and v.

See also:

remove_edges_from
remove a collection of edges

Examples

```
>>> G = nx.Graph()  # or DiGraph, etc
>>> nx.add_path(G, [0, 1, 2, 3])
>>> G.remove_edge(0, 1)
>>> e = (1, 2)
>>> G.remove_edge(*e)  # unpacks e from an edge tuple
>>> e = (2, 3, {"weight": 7})  # an edge with attribute data
>>> G.remove_edge(*e[:2])  # select first part of edge tuple
```

DiGraph.remove_edges_from

DiGraph.**remove_edges_from**(*ebunch*)

Remove all edges specified in ebunch.

Parameters

ebunch: list or container of edge tuples
Each edge given in the list or container will be removed from the graph. The edges can be:

- 2-tuples (u, v) edge between u and v.
- 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)
```

DiGraph.update

DiGraph.**update**(*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.

See also:

add_edges_from
: add multiple edges to a graph

add_nodes_from
: add multiple nodes to a graph

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}}
>>> e = [
...     (u, v, {"weight": d})
...     for u, nbrs in adj.items()
...     for v, d in nbrs.items()
... ]
>>> DG.update(edges=e, nodes=adj)
```

```
>>> # dict-of-dict-of-dict
>>> adj = {1: {2: {"weight": 1.3}, 3: {"color": 0.7, "weight": 1.2}}}
>>> e = [
...     (u, v, {"weight": d})
...     for u, nbrs in adj.items()
...     for v, d in nbrs.items()
... ]
>>> DG.update(edges=e, nodes=adj)
```

```
>>> # predecessor adjacency (dict-of-set)
>>> pred = {1: {2, 3}, 2: {3}, 3: {3}}
>>> e = [(v, u) for u, nbrs in pred.items() for v in nbrs]
```

```
>>> # MultiGraph dict-of-dict-of-dict-of-attribute
>>> MDG = nx.MultiDiGraph()
>>> adj = {
...     1: {2: {0: {"weight": 1.3}, 1: {"weight": 1.2}}},
...     3: {2: {0: {"weight": 0.7}}},
... }
>>> e = [
...     (u, v, ekey, d)
...     for u, nbrs in adj.items()
...     for v, keydict in nbrs.items()
...     for ekey, d in keydict.items()
... ]
>>> MDG.update(edges=e)
```

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)
```

DiGraph.clear

DiGraph.**clear**()

Remove all nodes and edges from the graph.

This also removes the name, and all graph, node, and edge attributes.

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.clear()
>>> list(G.nodes)
[]
>>> list(G.edges)
[]
```

DiGraph.clear_edges

DiGraph.**clear_edges**()

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)
[]
```

Reporting nodes edges and neighbors

DiGraph.nodes	A NodeView of the Graph as G.nodes or G.nodes().
DiGraph.__iter__()	Iterate over the nodes.
DiGraph.has_node(n)	Returns True if the graph contains the node n.
DiGraph.__contains__(n)	Returns True if n is a node, False otherwise.
DiGraph.edges	An OutEdgeView of the DiGraph as G.edges or G.edges().
DiGraph.out_edges	An OutEdgeView of the DiGraph as G.edges or G.edges().
DiGraph.in_edges	A view of the in edges of the graph as G.in_edges or G.in_edges().
DiGraph.has_edge(u, v)	Returns True if the edge (u, v) is in the graph.
DiGraph.get_edge_data(u, v[, default])	Returns the attribute dictionary associated with edge (u, v).
DiGraph.neighbors(n)	Returns an iterator over successor nodes of n.
DiGraph.adj	Graph adjacency object holding the neighbors of each node.
DiGraph.__getitem__(n)	Returns a dict of neighbors of node n.
DiGraph.successors(n)	Returns an iterator over successor nodes of n.
DiGraph.succ	Graph adjacency object holding the successors of each node.
DiGraph.predecessors(n)	Returns an iterator over predecessor nodes of n.
DiGraph.pred	Graph adjacency object holding the predecessors of each node.
DiGraph.adjacency()	Returns an iterator over (node, adjacency dict) tuples for all nodes.
DiGraph.nbunch_iter([nbunch])	Returns an iterator over nodes contained in nbunch that are also in the graph.

DiGraph.nodes

property DiGraph.**nodes**

A NodeView of the Graph as G.nodes or G.nodes().

Can be used as `G.nodes` for data lookup and for set-like operations. Can also be used as `G.nodes(data='color', default=None)` to return a NodeDataView which reports specific node data but no set operations. It presents a dict-like interface as well with `G.nodes.items()` iterating over `(node, nodedata)` 2-tuples and `G.nodes[3]['foo']` providing the value of the `foo` attribute for node `3`. In addition, a view `G.nodes.data('foo')` provides a dict-like interface to the `foo` attribute of each node. `G.nodes.data('foo', default=1)` provides a default for nodes that do not have attribute `foo`.

Parameters

data
[string or bool, optional (default=False)] The node attribute returned in 2-tuple (n, ddict[data]). If True, return entire node attribute dict as (n, ddict). If False, return just the nodes n.

default
[value, optional (default=None)] Value used for nodes that don't have the requested attribute. Only relevant if data is not True or False.

Returns

NodeView
Allows set-like operations over the nodes as well as node attribute dict lookup and calling to get a NodeDataView. A NodeDataView iterates over `(n, data)` and has no set operations. A NodeView iterates over `n` and includes set operations.

When called, if data is False, an iterator over nodes. Otherwise an iterator of 2-tuples (node, attribute value) where the attribute is specified in `data`. If data is True then the attribute becomes the entire data dictionary.

Notes

If your node data is not needed, it is simpler and equivalent to use the expression `for n in G`, or `list(G)`.

Examples

There are two simple ways of getting a list of all nodes in the graph:

```
>>> G = nx.path_graph(3)
>>> list(G.nodes)
[0, 1, 2]
>>> list(G)
[0, 1, 2]
```

To get the node data along with the nodes:

```
>>> G.add_node(1, time="5pm")
>>> G.nodes[0]["foo"] = "bar"
>>> list(G.nodes(data=True))
[(0, {'foo': 'bar'}), (1, {'time': '5pm'}), (2, {})]
>>> list(G.nodes.data())
[(0, {'foo': 'bar'}), (1, {'time': '5pm'}), (2, {})]
```

```
>>> list(G.nodes(data="foo"))
[(0, 'bar'), (1, None), (2, None)]
>>> list(G.nodes.data("foo"))
[(0, 'bar'), (1, None), (2, None)]
```

```
>>> list(G.nodes(data="time"))
[(0, None), (1, '5pm'), (2, None)]
>>> list(G.nodes.data("time"))
[(0, None), (1, '5pm'), (2, None)]
```

```
>>> list(G.nodes(data="time", default="Not Available"))
[(0, 'Not Available'), (1, '5pm'), (2, 'Not Available')]
>>> list(G.nodes.data("time", default="Not Available"))
[(0, 'Not Available'), (1, '5pm'), (2, 'Not Available')]
```

If some of your nodes have an attribute and the rest are assumed to have a default attribute value you can create a dictionary from node/attribute pairs using the `default` keyword argument to guarantee the value is never None:

```
>>> G = nx.Graph()
>>> G.add_node(0)
>>> G.add_node(1, weight=2)
>>> G.add_node(2, weight=3)
>>> dict(G.nodes(data="weight", default=1))
{0: 1, 1: 2, 2: 3}
```

DiGraph.__iter__

DiGraph.**__iter__**()

Iterate over the nodes. Use: 'for n in G'.

Returns

niter
[iterator] An iterator over all nodes in the graph.

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> [n for n in G]
[0, 1, 2, 3]
>>> list(G)
[0, 1, 2, 3]
```

DiGraph.has_node

DiGraph.**has_node**(*n*)

Returns True if the graph contains the node n.

Identical to `n in G`

Parameters

n
[node]

Examples

```
>>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.has_node(0)
True
```

It is more readable and simpler to use

```
>>> 0 in G
True
```

DiGraph.__contains__

DiGraph.**__contains__**(*n*)

Returns True if n is a node, False otherwise. Use: 'n in G'.

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> 1 in G
True
```

DiGraph.edges

property DiGraph.**edges**

An OutEdgeView of the DiGraph as G.edges or G.edges().

edges(self, nbunch=None, data=False, default=None)

The OutEdgeView provides set-like operations on the edge-tuples as well as edge attribute lookup. When called, it also provides an EdgeDataView object which allows control of access to edge attributes (but does not provide set-like operations). Hence, `G.edges[u, v]['color']` provides the value of the color attribute for edge `(u, v)` while `for (u, v, c) in G.edges.data('color', default='red'):` iterates through all the edges yielding the color attribute with default `'red'` if no color attribute exists.

Parameters

nbunch
[single node, container, or all nodes (default= all nodes)] The view will only report edges from 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).

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

edges
[OutEdgeView] 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']`.

See also:

in_edges, *out_edges*

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.DiGraph()  # or MultiDiGraph, etc
>>> nx.add_path(G, [0, 1, 2])
>>> 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)
OutEdgeDataView([(0, 1, {}), (1, 2, {}), (2, 3, {'weight': 5})])
>>> G.edges.data("weight", default=1)
OutEdgeDataView([(0, 1, 1), (1, 2, 1), (2, 3, 5)])
>>> G.edges([0, 2])  # only edges originating from these nodes
OutEdgeDataView([(0, 1), (2, 3)])
>>> G.edges(0)  # only edges from node 0
OutEdgeDataView([(0, 1)])
```

DiGraph.out_edges

property DiGraph.**out_edges**

An OutEdgeView of the DiGraph as G.edges or G.edges().

edges(self, nbunch=None, data=False, default=None)

The OutEdgeView provides set-like operations on the edge-tuples as well as edge attribute lookup. When called, it also provides an EdgeDataView object which allows control of access to edge attributes (but does not provide set-like operations). Hence, `G.edges[u, v]['color']` provides the value of the color attribute for edge `(u, v)` while `for (u, v, c) in G.edges.data('color', default='red'):` iterates through all the edges yielding the color attribute with default `'red'` if no color attribute exists.

Parameters

nbunch
[single node, container, or all nodes (default= all nodes)] The view will only report edges from 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).

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

edges
[OutEdgeView] 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']`.

See also:

in_edges, *out_edges*

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.DiGraph()  # or MultiDiGraph, etc
>>> nx.add_path(G, [0, 1, 2])
>>> 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)
OutEdgeDataView([(0, 1, {}), (1, 2, {}), (2, 3, {'weight': 5})])
>>> G.edges.data("weight", default=1)
OutEdgeDataView([(0, 1, 1), (1, 2, 1), (2, 3, 5)])
>>> G.edges([0, 2])  # only edges originating from these nodes
OutEdgeDataView([(0, 1), (2, 3)])
>>> G.edges(0)  # only edges from node 0
OutEdgeDataView([(0, 1)])
```

DiGraph.in_edges

property DiGraph.**in_edges**

A view of the in edges of the graph as G.in_edges or G.in_edges().

in_edges(self, nbunch=None, data=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).

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
[InEdgeView or InEdgeDataView] 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']`.

See also:

edges

Examples

```
>>> G = nx.DiGraph()
>>> G.add_edge(1, 2, color='blue')
>>> G.in_edges()
InEdgeView([(1, 2)])
>>> G.in_edges(nbunch=2)
InEdgeDataView([(1, 2)])
```

DiGraph.has_edge

DiGraph.**has_edge**(*u*, *v*)

Returns True if the edge (u, v) is in the graph.

This is the same as `v in G[u]` without KeyError exceptions.

Parameters

u, v
[nodes] Nodes can be, for example, strings or numbers. Nodes must be hashable (and not None) Python objects.

Returns

edge_ind
[bool] True if edge is in the graph, False otherwise.

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.has_edge(0, 1)  # using two nodes
True
>>> e = (0, 1)
>>> G.has_edge(*e)  #  e is a 2-tuple (u, v)
True
>>> e = (0, 1, {"weight": 7})
>>> G.has_edge(*e[:2])  # e is a 3-tuple (u, v, data_dictionary)
True
```

The following syntax are equivalent:

```
>>> G.has_edge(0, 1)
True
>>> 1 in G[0]  # though this gives KeyError if 0 not in G
True
```

DiGraph.get_edge_data

DiGraph.**get_edge_data**(*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
```

DiGraph.neighbors

DiGraph.**neighbors**(*n*)

Returns an iterator over successor nodes of n.

A successor of n is a node m such that there exists a directed edge from n to m.

Parameters

n
[node] A node in the graph

Raises

NetworkXError
If n is not in the graph.

See also:

predecessors

Notes

neighbors() and successors() are the same.

DiGraph.adj

property DiGraph.**adj**

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.

DiGraph.__getitem__

DiGraph.**__getitem__**(*n*)

Returns a dict of neighbors of node n. Use: 'G[n]'.

Parameters

n
[node] A node in the graph.

Returns

adj_dict
[dictionary] The adjacency dictionary for nodes connected to n.

Notes

G[n] is the same as G.adj[n] and similar to G.neighbors(n) (which is an iterator over G.adj[n])

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G[0]
AtlasView({1: {}})
```

DiGraph.successors

DiGraph.**successors**(*n*)

Returns an iterator over successor nodes of n.

A successor of n is a node m such that there exists a directed edge from n to m.

Parameters

n
[node] A node in the graph

Raises

NetworkXError
If n is not in the graph.

See also:

predecessors

Notes

neighbors() and successors() are the same.

DiGraph.succ

property DiGraph.**succ**

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`.

DiGraph.predecessors

DiGraph.**predecessors**(*n*)

Returns an iterator over predecessor nodes of n.

A predecessor of n is a node m such that there exists a directed edge from m to n.

Parameters

n
[node] A node in the graph

Raises

NetworkXError
If n is not in the graph.

See also:

successors

DiGraph.pred

property DiGraph.**pred**

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.

DiGraph.adjacency

DiGraph.**adjacency**()

Returns an iterator over (node, adjacency dict) tuples for all nodes.

For directed graphs, only outgoing neighbors/adjacencies are included.

Returns

adj_iter
[iterator] An iterator over (node, adjacency dictionary) for all nodes in the graph.

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> [(n, nbrdict) for n, nbrdict in G.adjacency()]
[(0, {1: {}}), (1, {0: {}, 2: {}}), (2, {1: {}, 3: {}}), (3, {2: {}})]
```

DiGraph.nbunch_iter

DiGraph.**nbunch_iter**(*nbunch=None*)

Returns an iterator over nodes contained in nbunch that are also in the graph.

The nodes in nbunch are checked for membership in the graph and if not are silently ignored.

Parameters

nbunch
[single node, container, or all nodes (default= all nodes)] The view will only report edges incident to these nodes.

Returns

niter
[iterator] An iterator over nodes in nbunch that are also in the graph. If nbunch is None, iterate over all nodes in the graph.

Raises

NetworkXError
If nbunch is not a node or sequence of nodes. If a node in nbunch is not hashable.

See also:

Graph.__iter__

Notes

When nbunch is an iterator, the returned iterator yields values directly from nbunch, becoming exhausted when nbunch is exhausted.

To test whether nbunch is a single node, one can use "if nbunch in self:", even after processing with this routine.

If nbunch is not a node or a (possibly empty) sequence/iterator or None, a *NetworkXError* is raised. Also, if any object in nbunch is not hashable, a *NetworkXError* is raised.

Counting nodes edges and neighbors

DiGraph.order()	Returns the number of nodes in the graph.
DiGraph.number_of_nodes()	Returns the number of nodes in the graph.
DiGraph.__len__()	Returns the number of nodes in the graph.
DiGraph.degree	A DegreeView for the Graph as G.degree or G.degree().
DiGraph.in_degree	An InDegreeView for (node, in_degree) or in_degree for single node.
DiGraph.out_degree	An OutDegreeView for (node, out_degree)
DiGraph.size([weight])	Returns the number of edges or total of all edge weights.
DiGraph.number_of_edges([u, v])	Returns the number of edges between two nodes.

DiGraph.order

DiGraph.**order**()

Returns the number of nodes in the graph.

Returns

nnodes
[int] The number of nodes in the graph.

See also:

number_of_nodes
identical method

__len__
identical method

Examples

```
>>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.order()
3
```

DiGraph.number_of_nodes

DiGraph.**number_of_nodes**()

Returns the number of nodes in the graph.

Returns

nnodes
[int] The number of nodes in the graph.

See also:

order
identical method

__len__
identical method

Examples

```
>>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.number_of_nodes()
3
```

DiGraph.__len__

DiGraph.**__len__**()

Returns the number of nodes in the graph. Use: 'len(G)'.

Returns

nnodes
[int] The number of nodes in the graph.

See also:

number_of_nodes
identical method

order
identical method

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> len(G)
4
```

DiGraph.degree

property DiGraph.**degree**

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

DiDegreeView or int
If multiple nodes are requested (the default), returns a `DiDegreeView` mapping nodes to their degree. If a single node is requested, returns the degree of the node as an integer.

See also:

in_degree, *out_degree*

Examples

```
>>> G = nx.DiGraph()  # or 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)]
```

DiGraph.in_degree

property DiGraph.**in_degree**

An InDegreeView for (node, in_degree) or in_degree for single node.

The node in_degree is the number of edges pointing to the node. The weighted node degree is the sum of the edge weights for edges incident to that node.

This object provides an iteration over (node, in_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

If a single node is requested
deg
[int] In-degree of the node

OR if multiple nodes are requested
nd_iter
[iterator] The iterator returns two-tuples of (node, in-degree).

See also:

degree, *out_degree*

Examples

```
>>> G = nx.DiGraph()
>>> nx.add_path(G, [0, 1, 2, 3])
>>> G.in_degree(0)  # node 0 with degree 0
0
>>> list(G.in_degree([0, 1, 2]))
[(0, 0), (1, 1), (2, 1)]
```

DiGraph.out_degree

property DiGraph.**out_degree**

An OutDegreeView for (node, out_degree)

The node out_degree is the number of edges pointing out of the node. The weighted node degree is the sum of the edge weights for edges incident to that node.

This object provides an iterator over (node, out_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

If a single node is requested
deg
[int] Out-degree of the node

OR if multiple nodes are requested
nd_iter
[iterator] The iterator returns two-tuples of (node, out-degree).

See also:

degree, *in_degree*

Examples

```
>>> G = nx.DiGraph()
>>> nx.add_path(G, [0, 1, 2, 3])
>>> G.out_degree(0)  # node 0 with degree 1
1
>>> list(G.out_degree([0, 1, 2]))
[(0, 1), (1, 1), (2, 1)]
```

DiGraph.size

DiGraph.**size**(*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
```

DiGraph.number_of_edges

DiGraph.**number_of_edges**(*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
```

Making copies and subgraphs

DiGraph.copy([as_view])	Returns a copy of the graph.
DiGraph.to_undirected([reciprocal, as_view])	Returns an undirected representation of the digraph.
DiGraph.to_directed([as_view])	Returns a directed representation of the graph.
DiGraph.subgraph(nodes)	Returns a SubGraph view of the subgraph induced on `nodes`.
DiGraph.edge_subgraph(edges)	Returns the subgraph induced by the specified edges.
DiGraph.reverse([copy])	Returns the reverse of the graph.

DiGraph.copy

DiGraph.**copy**(*as_view=False*)

Returns a copy of the graph.

The copy method by default returns an independent shallow copy of the graph and attributes. That is, if an attribute is a container, that container is shared by the original an the copy. Use Python's `copy.deepcopy` for new containers.

If `as_view` is True then a view is returned instead of a copy.

Parameters

as_view
[bool, optional (default=False)] If True, the returned graph-view provides a read-only view of the original graph without actually copying any data.

Returns

G
[Graph] A copy of the graph.

See also:

to_directed
return a directed copy of the graph.

Notes

All copies reproduce the graph structure, but data attributes may be handled in different ways. There are four types of copies of a graph that people might want.

Deepcopy – A "deepcopy" copies the graph structure as well as all data attributes and any objects they might contain. The entire graph object is new so that changes in the copy do not affect the original object. (see Python's copy.deepcopy)

Data Reference (Shallow) – For a shallow copy the graph structure is copied but the edge, node and graph attribute dicts are references to those in the original graph. This saves time and memory but could cause confusion if you change an attribute in one graph and it changes the attribute in the other. NetworkX does not provide this level of shallow copy.

Independent Shallow – This copy creates new independent attribute dicts and then does a shallow copy of the attributes. That is, any attributes that are containers are shared between the new graph and the original. This is exactly what `dict.copy()` provides. You can obtain this style copy using:

```
>>> G = nx.path_graph(5)
>>> H = G.copy()
>>> H = G.copy(as_view=False)
>>> H = nx.Graph(G)
>>> H = G.__class__(G)
```

Fresh Data – For fresh data, the graph structure is copied while new empty data attribute dicts are created. The resulting graph is independent of the original and it has no edge, node or graph attributes. Fresh copies are not enabled. Instead use:

```
>>> H = G.__class__()
>>> H.add_nodes_from(G)
>>> H.add_edges_from(G.edges)
```

View – Inspired by dict-views, graph-views act like read-only versions of the original graph, providing a copy of the original structure without requiring any memory for copying the information.

See the Python copy module for more information on shallow and deep copies, https://docs.python.org/3/library/copy.html.

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> H = G.copy()
```

DiGraph.to_undirected

DiGraph.**to_undirected**(*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)]
```

DiGraph.to_directed

DiGraph.**to_directed**(*as_view=False*)

Returns a directed representation of the graph.

Returns

G

[DiGraph] A directed graph with the same name, same nodes, and with each edge (u, v, data) replaced by two directed edges (u, v, data) and (v, u, data).

Notes

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 D=DiGraph(G) 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 Graph to use dict-like objects in the data structure, those changes do not transfer to the DiGraph created by this method.

Examples

```
>>> G = nx.Graph()  # or MultiGraph, etc
>>> G.add_edge(0, 1)
>>> H = G.to_directed()
>>> list(H.edges)
[(0, 1), (1, 0)]
```

If already directed, return a (deep) copy

```
>>> G = nx.DiGraph()  # or MultiDiGraph, etc
>>> G.add_edge(0, 1)
>>> H = G.to_directed()
>>> list(H.edges)
[(0, 1)]
```

DiGraph.subgraph

DiGraph.**subgraph**(*nodes*)

Returns a SubGraph view of the subgraph induced on *nodes*.

The induced subgraph of the graph contains the nodes in *nodes* and the edges between those nodes.

Parameters

nodes
[list, iterable] A container of nodes which will be iterated through once.

Returns

G
[SubGraph View] A subgraph view of the graph. The graph structure cannot be changed but node/edge attributes can and are shared with the original graph.

Notes

The graph, edge and node attributes are shared with the original graph. Changes to the graph structure is ruled out by the view, but changes to attributes are reflected in the original graph.

To create a subgraph with its own copy of the edge/node attributes use: G.subgraph(nodes).copy()

For an inplace reduction of a graph to a subgraph you can remove nodes: G.remove_nodes_from([n for n in G if n not in set(nodes)])

Subgraph views are sometimes NOT what you want. In most cases where you want to do more than simply look at the induced edges, it makes more sense to just create the subgraph as its own graph with code like:

```
# Create a subgraph SG based on a (possibly multigraph) G
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)]
```

DiGraph.edge_subgraph

DiGraph.**edge_subgraph**(*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)]
```

DiGraph.reverse

DiGraph.**reverse**(*copy=True*)

Returns the reverse of the graph.

The reverse is a graph with the same nodes and edges but with the directions of the edges reversed.

Parameters

copy
[bool optional (default=True)] If True, return a new DiGraph holding the reversed edges. If False, the reverse graph is created using a view of the original graph.

2.2.3 MultiGraph—Undirected graphs with self loops and parallel edges

Overview

class MultiGraph(*incoming_graph_data=None*, *multigraph_input=None*, ***attr*)

An undirected graph class that can store multiedges.

Multiedges are multiple edges between two nodes. Each edge can hold optional data or attributes.

A MultiGraph holds undirected edges. Self loops are allowed.

Nodes can be arbitrary (hashable) Python objects with optional key/value attributes. By convention `None` is not used as a node.

Edges are represented as links between nodes with optional key/value attributes, in a MultiGraph each edge has a key to distinguish between multiple edges that have the same source and destination nodes.

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 any format that is supported by the to_networkx_graph() function, currently including edge list, dict of dicts, dict of lists, NetworkX graph, 2D NumPy array, SciPy sparse array, or 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, `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:

Graph
DiGraph
MultiDiGraph

Examples

Create an empty graph structure (a "null graph") with no nodes and no edges.

```
>>> G = nx.MultiGraph()
```

G can be grown in several ways.

Nodes:

Add one node at a time:

```
>>> G.add_node(1)
```

Add the nodes from any container (a list, dict, set or even the lines from a file or the nodes from another graph).

```
>>> G.add_nodes_from([2, 3])
>>> G.add_nodes_from(range(100, 110))
>>> H = nx.path_graph(10)
>>> G.add_nodes_from(H)
```

In addition to strings and integers any hashable Python object (except None) can represent a node, e.g. a customized node object, or even another Graph.

```
>>> G.add_node(H)
```

Edges:

G can also be grown by adding edges.

Add one edge,

```
>>> key = G.add_edge(1, 2)
```

a list of edges,

```
>>> keys = G.add_edges_from([(1, 2), (1, 3)])
```

or a collection of edges,

```
>>> keys = G.add_edges_from(H.edges)
```

If some edges connect nodes not yet in the graph, the nodes are added automatically. If an edge already exists, an additional edge is created and stored using a key to identify the edge. By default the key is the lowest unused integer.

```
>>> keys = G.add_edges_from([(4, 5, {"route": 28}), (4, 5, {"route": 37})])
>>> G[4]
AdjacencyView({3: {0: {}}, 5: {0: {}, 1: {'route': 28}, 2: {'route': 37}}})
```

Attributes:

Each graph, node, and edge can hold key/value attribute pairs in an associated attribute dictionary (the keys must be hashable). By default these are empty, but can be added or changed using add_edge, add_node or direct manipulation of the attribute dictionaries named graph, node and edge respectively.

```
>>> G = nx.MultiGraph(day="Friday")
>>> G.graph
{'day': 'Friday'}
```

Add node attributes using add_node(), add_nodes_from() or G.nodes

```
>>> G.add_node(1, time="5pm")
>>> G.add_nodes_from([3], time="2pm")
>>> G.nodes[1]
{'time': '5pm'}
>>> G.nodes[1]["room"] = 714
>>> del G.nodes[1]["room"]  # remove attribute
>>> list(G.nodes(data=True))
[(1, {'time': '5pm'}), (3, {'time': '2pm'})]
```

Add edge attributes using add_edge(), add_edges_from(), subscript notation, or G.edges.

```
>>> key = G.add_edge(1, 2, weight=4.7)
>>> keys = G.add_edges_from([(3, 4), (4, 5)], color="red")
>>> keys = G.add_edges_from([(1, 2, {"color": "blue"}), (2, 3, {"weight": 8})])
>>> G[1][2][0]["weight"] = 4.7
>>> G.edges[1, 2, 0]["weight"] = 4
```

Warning: we protect the graph data structure by making `G.edges[1, 2, 0]` a read-only dict-like structure. However, you can assign to attributes in e.g. `G.edges[1, 2, 0]`. Thus, use 2 sets of brackets to add/change data attributes: `G.edges[1, 2, 0]['weight'] = 4`.

Shortcuts:

Many common graph features allow python syntax to speed reporting.

```
>>> 1 in G  # check if node in graph
True
>>> [n for n in G if n < 3]  # iterate through nodes
[1, 2]
>>> len(G)  # number of nodes in graph
5
>>> G[1]  # adjacency dict-like view mapping neighbor -> edge key -> edge␣
↪attributes
AdjacencyView({2: {0: {'weight': 4}, 1: {'color': 'blue'}}})
```

Often the best way to traverse all edges of a graph is via the neighbors. The neighbors are reported as an adjacency-dict `G.adj` or `G.adjacency()`.

```
>>> for n, nbrsdict in G.adjacency():
...     for nbr, keydict in nbrsdict.items():
...         for key, eattr in keydict.items():
...             if "weight" in eattr:
...                 # Do something useful with the edges
...                 pass
```

But the edges() method is often more convenient:

```
>>> for u, v, keys, weight in G.edges(data="weight", keys=True):
...     if weight is not None:
...         # Do something useful with the edges
...         pass
```

Reporting:

Simple graph information is obtained using methods and object-attributes. Reporting usually provides views instead of containers to reduce memory usage. The views update as the graph is updated similarly to dict-views. The objects *nodes*, *edges* and *adj* provide access to data attributes via lookup (e.g. `nodes[n]`, `edges[u, v, k]`, `adj[u][v]`) and iteration (e.g. `nodes.items()`, `nodes.data('color')`, `nodes.data('color', default='blue')` and similarly for *edges*) Views exist for *nodes*, *edges*, *neighbors()*/*adj* and *degree*.

For details on these and other miscellaneous methods, see below.

Subclasses (Advanced):

The MultiGraph class uses a dict-of-dict-of-dict-of-dict data structure. The outer dict (node_dict) holds adjacency information keyed by node. The next dict (adjlist_dict) represents the adjacency information and holds edge_key dicts keyed by neighbor. The edge_key dict holds each edge_attr dict keyed by edge key. The inner dict (edge_attr_dict) represents the edge data and holds edge attribute values keyed by attribute names.

Each of these four dicts in the dict-of-dict-of-dict-of-dict structure can be replaced 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. 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()
```

(continues on next page)

(continued from previous page)

```
>>> G.add_edge(2, 1)
>>> G[2][1]
{'weight': 1}
>>> G.add_edge(2, 2)
>>> G[2][1] is G[2][2]
True
```

Methods

Adding and removing nodes and edges

MultiGraph.__init__([incoming_graph_data, ...])	Initialize a graph with edges, name, or graph attributes.
MultiGraph.add_node(node_for_adding, **attr)	Add a single node `node_for_adding` and update node attributes.
MultiGraph.add_nodes_from(nodes_for_adding, ...)	Add multiple nodes.
MultiGraph.remove_node(n)	Remove node n.
MultiGraph.remove_nodes_from(nodes)	Remove multiple nodes.
MultiGraph.add_edge(u_for_edge, v_for_edge)	Add an edge between u and v.
MultiGraph.add_edges_from(ebunch_to_add, **attr)	Add all the edges in ebunch_to_add.
MultiGraph.add_weighted_edges_from(ebunch_to_add)	Add weighted edges in `ebunch_to_add` with specified weight attr
MultiGraph.new_edge_key(u, v)	Returns an unused key for edges between nodes `u` and `v`.
MultiGraph.remove_edge(u, v[, key])	Remove an edge between u and v.
MultiGraph.remove_edges_from(ebunch)	Remove all edges specified in ebunch.
MultiGraph.update([edges, nodes])	Update the graph using nodes/edges/graphs as input.
MultiGraph.clear()	Remove all nodes and edges from the graph.
MultiGraph.clear_edges()	Remove all edges from the graph without altering nodes.

MultiGraph.__init__

`MultiGraph.__init__`(*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, `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'}
```

MultiGraph.add_node

MultiGraph.**add_node**(*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

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.

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))
```

MultiGraph.add_nodes_from

MultiGraph.**add_nodes_from**(*nodes_for_adding*, ***attr*)

Add multiple nodes.

Parameters

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))
```

MultiGraph.remove_node

MultiGraph.**remove_node**(*n*)

Remove node n.

Removes the node n and all adjacent edges. Attempting to remove a non-existent node will raise an exception.

Parameters

n
[node] A node in the graph

Raises

NetworkXError
If n is not in the graph.

See also:

remove_nodes_from

Examples

```
>>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> list(G.edges)
[(0, 1), (1, 2)]
>>> G.remove_node(1)
>>> list(G.edges)
[]
```

MultiGraph.remove_nodes_from

MultiGraph.**remove_nodes_from**(*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`.

Examples

```
>>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> e = list(G.nodes)
>>> e
[0, 1, 2]
>>> G.remove_nodes_from(e)
>>> list(G.nodes)
[]
```

Evaluate an iterator over a graph if using it to modify the same graph

```
>>> G = nx.Graph([(0, 1), (1, 2), (3, 4)])
>>> # this command will fail, as the graph's dict is modified during iteration
>>> # G.remove_nodes_from(n for n in G.nodes if n < 2)
>>> # this command will work, since the dictionary underlying graph is not␣
↪modified
>>> G.remove_nodes_from(list(n for n in G.nodes if n < 2))
```

MultiGraph.add_edge

MultiGraph.**add_edge**(*u_for_edge*, *v_for_edge*, *key=None*, ***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_for_edge, v_for_edge
: [nodes] Nodes can be, for example, strings or numbers. 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 each add an additional edge e=(1, 2) to graph G:

```
>>> G = nx.MultiGraph()
>>> e = (1, 2)
>>> ekey = 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:

```
>>> ekey = G.add_edge(1, 2, weight=3)
>>> ekey = G.add_edge(1, 2, key=0, weight=4)  # update data for key=0
>>> ekey = 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})
```

MultiGraph.add_edges_from

MultiGraph.**add_edges_from**(*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 can be:

- 2-tuples (u, v) or
- 3-tuples (u, v, d) for an edge data dict d, or
- 3-tuples (u, v, k) for not iterable key k, or
- 4-tuples (u, v, k, d) for an edge with data and key k

attr
[keyword arguments, optional] Edge data (or labels or objects) can be assigned using keyword arguments.

Returns

A list of edge keys assigned to the edges in ebunch.

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))
```

MultiGraph.add_weighted_edges_from

MultiGraph.**add_weighted_edges_from**(*ebunch_to_add*, *weight='weight'*, ***attr*)

Add weighted edges in `ebunch_to_add` with specified weight attr

Parameters

ebunch_to_add
: [container of edges] Each edge given in the list or container will be added to the graph. The edges must be given as 3-tuples (u, v, w) where w is a number.

weight
: [string, optional (default= 'weight')] The attribute name for the edge weights to be added.

attr
: [keyword arguments, optional (default= no attributes)] Edge attributes to add/update for all edges.

See also:

add_edge
: add a single edge

add_edges_from
: add multiple edges

Notes

Adding the same edge twice for Graph/DiGraph simply updates the edge data. For MultiGraph/MultiDiGraph, duplicate edges are stored.

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_weighted_edges_from`.

Examples

```
>>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_weighted_edges_from([(0, 1, 3.0), (1, 2, 7.5)])
```

Evaluate an iterator over edges before passing it

```
>>> G = nx.Graph([(1, 2), (2, 3), (3, 4)])
>>> weight = 0.1
>>> # Grow graph by one new node, adding edges to all existing nodes.
>>> # wrong way - will raise RuntimeError
>>> # G.add_weighted_edges_from(((5, n, weight) for n in G.nodes))
>>> # correct way - note that there will be no self-edge for node 5
>>> G.add_weighted_edges_from(list((5, n, weight) for n in G.nodes))
```

MultiGraph.new_edge_key

`MultiGraph.`**`new_edge_key`**`(u, v)`

Returns an unused key for edges between nodes `u` and `v`.

The nodes `u` and `v` do not need to be already in the graph.

Parameters

u, v
[nodes]

Returns

key
[int]

Notes

In the standard MultiGraph class the new key is the number of existing edges between `u` and `v` (increased if necessary to ensure unused). The first edge will have key 0, then 1, etc. If an edge is removed further new_edge_keys may not be in this order.

MultiGraph.remove_edge

MultiGraph.**remove_edge**(*u*, *v*, *key=None*)

Remove an edge between u and v.

Parameters

u, v
[nodes] Remove an edge between nodes u and v.

key
[hashable identifier, optional (default=None)] Used to distinguish multiple edges between a pair of nodes. If None, remove a single edge between u and v. If there are multiple edges, removes the last edge added in terms of insertion order.

Raises

NetworkXError
If there is not an edge between u and v, or if there is no edge with the specified key.

See also:

remove_edges_from
remove a collection of edges

Examples

```
>>> G = nx.MultiGraph()
>>> nx.add_path(G, [0, 1, 2, 3])
>>> G.remove_edge(0, 1)
>>> e = (1, 2)
>>> G.remove_edge(*e)  # unpacks e from an edge tuple
```

For multiple edges

```
>>> G = nx.MultiGraph()  # or MultiDiGraph, etc
>>> G.add_edges_from([(1, 2), (1, 2), (1, 2)])  # key_list returned
[0, 1, 2]
```

When `key=None` (the default), edges are removed in the opposite order that they were added:

```
>>> G.remove_edge(1, 2)
>>> G.edges(keys=True)
MultiEdgeView([(1, 2, 0), (1, 2, 1)])
>>> G.remove_edge(2, 1)  # edges are not directed
>>> G.edges(keys=True)
MultiEdgeView([(1, 2, 0)])
```

For edges with keys

```
>>> G = nx.MultiGraph()
>>> G.add_edge(1, 2, key="first")
'first'
>>> G.add_edge(1, 2, key="second")
'second'
>>> G.remove_edge(1, 2, key="first")
```

(continues on next page)

(continued from previous page)

```
>>> G.edges(keys=True)
MultiEdgeView([(1, 2, 'second')])
```

MultiGraph.remove_edges_from

MultiGraph.**remove_edges_from**(*ebunch*)

Remove all edges specified in ebunch.

Parameters

ebunch: list or container of edge tuples
Each edge given in the list or container will be removed from the graph. The edges can be:

- 2-tuples (u, v) A single edge between u and v is removed.
- 3-tuples (u, v, key) The edge identified by key is removed.
- 4-tuples (u, v, key, data) where data 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)
```

Removing multiple copies of edges

```
>>> G = nx.MultiGraph()
>>> keys = G.add_edges_from([(1, 2), (1, 2), (1, 2)])
>>> G.remove_edges_from([(1, 2), (2, 1)])  # edges aren't directed
>>> list(G.edges())
[(1, 2)]
>>> G.remove_edges_from([(1, 2), (1, 2)])  # silently ignore extra copy
>>> list(G.edges)  # now empty graph
[]
```

When the edge is a 2-tuple `(u, v)` but there are multiple edges between u and v in the graph, the most recent edge (in terms of insertion order) is removed.

```
>>> G = nx.MultiGraph()
>>> for key in ("x", "y", "a"):
...     k = G.add_edge(0, 1, key=key)
>>> G.edges(keys=True)
MultiEdgeView([(0, 1, 'x'), (0, 1, 'y'), (0, 1, 'a')])
```

(continues on next page)

(continued from previous page)

```
>>> G.remove_edges_from([(0, 1)])
>>> G.edges(keys=True)
MultiEdgeView([(0, 1, 'x'), (0, 1, 'y')])
```

MultiGraph.update

MultiGraph.**update**(*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.

See also:

add_edges_from
: add multiple edges to a graph

add_nodes_from
: add multiple nodes to a graph

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}}
>>> e = [
...     (u, v, {"weight": d})
```

(continues on next page)

(continued from previous page)

```
...     for u, nbrs in adj.items()
...     for v, d in nbrs.items()
... ]
>>> DG.update(edges=e, nodes=adj)
```

```
>>> # dict-of-dict-of-dict
>>> adj = {1: {2: {"weight": 1.3}, 3: {"color": 0.7, "weight": 1.2}}}
>>> e = [
...     (u, v, {"weight": d})
...     for u, nbrs in adj.items()
...     for v, d in nbrs.items()
... ]
>>> DG.update(edges=e, nodes=adj)
```

```
>>> # predecessor adjacency (dict-of-set)
>>> pred = {1: {2, 3}, 2: {3}, 3: {3}}
>>> e = [(v, u) for u, nbrs in pred.items() for v in nbrs]
```

```
>>> # MultiGraph dict-of-dict-of-dict-of-attribute
>>> MDG = nx.MultiDiGraph()
>>> adj = {
...     1: {2: {0: {"weight": 1.3}, 1: {"weight": 1.2}}},
...     3: {2: {0: {"weight": 0.7}}},
... }
>>> e = [
...     (u, v, ekey, d)
...     for u, nbrs in adj.items()
...     for v, keydict in nbrs.items()
...     for ekey, d in keydict.items()
... ]
>>> MDG.update(edges=e)
```

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)
```

MultiGraph.clear

MultiGraph.**clear**()

Remove all nodes and edges from the graph.

This also removes the name, and all graph, node, and edge attributes.

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.clear()
>>> list(G.nodes)
[]
>>> list(G.edges)
[]
```

MultiGraph.clear_edges

MultiGraph.**clear_edges**()

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)
[]
```

Reporting nodes edges and neighbors

MultiGraph.nodes	A NodeView of the Graph as G.nodes or G.nodes().
MultiGraph.__iter__()	Iterate over the nodes.
MultiGraph.has_node(n)	Returns True if the graph contains the node n.
MultiGraph.__contains__(n)	Returns True if n is a node, False otherwise.
MultiGraph.edges	Returns an iterator over the edges.
MultiGraph.has_edge(u, v[, key])	Returns True if the graph has an edge between nodes u and v.
MultiGraph.get_edge_data(u, v[, key, default])	Returns the attribute dictionary associated with edge (u, v, key).
MultiGraph.neighbors(n)	Returns an iterator over all neighbors of node n.
MultiGraph.adj	Graph adjacency object holding the neighbors of each node.
MultiGraph.__getitem__(n)	Returns a dict of neighbors of node n.
MultiGraph.adjacency()	Returns an iterator over (node, adjacency dict) tuples for all nodes.
MultiGraph.nbunch_iter([nbunch])	Returns an iterator over nodes contained in nbunch that are also in the graph.

MultiGraph.nodes

property MultiGraph.**nodes**

A NodeView of the Graph as G.nodes or G.nodes().

Can be used as `G.nodes` for data lookup and for set-like operations. Can also be used as `G.nodes(data='color', default=None)` to return a NodeDataView which reports specific node data but no set operations. It presents a dict-like interface as well with `G.nodes.items()` iterating over `(node, nodedata)` 2-tuples and `G.nodes[3]['foo']` providing the value of the `foo` attribute for node `3`. In addition, a view `G.nodes.data('foo')` provides a dict-like interface to the `foo` attribute of each node. `G.nodes.data('foo', default=1)` provides a default for nodes that do not have attribute `foo`.

Parameters

data
[string or bool, optional (default=False)] The node attribute returned in 2-tuple (n, ddict[data]). If True, return entire node attribute dict as (n, ddict). If False, return just the nodes n.

default
[value, optional (default=None)] Value used for nodes that don't have the requested attribute. Only relevant if data is not True or False.

Returns

NodeView
Allows set-like operations over the nodes as well as node attribute dict lookup and calling to get a NodeDataView. A NodeDataView iterates over `(n, data)` and has no set operations. A NodeView iterates over `n` and includes set operations.

When called, if data is False, an iterator over nodes. Otherwise an iterator of 2-tuples (node, attribute value) where the attribute is specified in `data`. If data is True then the attribute becomes the entire data dictionary.

Notes

If your node data is not needed, it is simpler and equivalent to use the expression `for n in G`, or `list(G)`.

Examples

There are two simple ways of getting a list of all nodes in the graph:

```
>>> G = nx.path_graph(3)
>>> list(G.nodes)
[0, 1, 2]
>>> list(G)
[0, 1, 2]
```

To get the node data along with the nodes:

```
>>> G.add_node(1, time="5pm")
>>> G.nodes[0]["foo"] = "bar"
>>> list(G.nodes(data=True))
[(0, {'foo': 'bar'}), (1, {'time': '5pm'}), (2, {})]
>>> list(G.nodes.data())
[(0, {'foo': 'bar'}), (1, {'time': '5pm'}), (2, {})]
```

```
>>> list(G.nodes(data="foo"))
[(0, 'bar'), (1, None), (2, None)]
>>> list(G.nodes.data("foo"))
[(0, 'bar'), (1, None), (2, None)]
```

```
>>> list(G.nodes(data="time"))
[(0, None), (1, '5pm'), (2, None)]
>>> list(G.nodes.data("time"))
[(0, None), (1, '5pm'), (2, None)]
```

```
>>> list(G.nodes(data="time", default="Not Available"))
[(0, 'Not Available'), (1, '5pm'), (2, 'Not Available')]
>>> list(G.nodes.data("time", default="Not Available"))
[(0, 'Not Available'), (1, '5pm'), (2, 'Not Available')]
```

If some of your nodes have an attribute and the rest are assumed to have a default attribute value you can create a dictionary from node/attribute pairs using the `default` keyword argument to guarantee the value is never None:

```
>>> G = nx.Graph()
>>> G.add_node(0)
>>> G.add_node(1, weight=2)
>>> G.add_node(2, weight=3)
>>> dict(G.nodes(data="weight", default=1))
{0: 1, 1: 2, 2: 3}
```

MultiGraph.__iter__

`MultiGraph.__iter__()`

Iterate over the nodes. Use: 'for n in G'.

Returns

niter
[iterator] An iterator over all nodes in the graph.

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> [n for n in G]
[0, 1, 2, 3]
>>> list(G)
[0, 1, 2, 3]
```

MultiGraph.has_node

MultiGraph.**has_node**(*n*)

Returns True if the graph contains the node n.

Identical to `n in G`

Parameters

n
[node]

Examples

```
>>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.has_node(0)
True
```

It is more readable and simpler to use

```
>>> 0 in G
True
```

MultiGraph.__contains__

MultiGraph.**__contains__**(*n*)

Returns True if n is a node, False otherwise. Use: 'n in G'.

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> 1 in G
True
```

MultiGraph.edges

property MultiGraph.**edges**

Returns an iterator over the edges.

edges(self, nbunch=None, data=False, keys=False, default=None)

The MultiEdgeView provides set-like operations on the edge-tuples as well as edge attribute lookup. When called, it also provides an EdgeDataView object which allows control of access to edge attributes (but does not provide set-like operations). Hence, `G.edges[u, v, k]['color']` provides the value of the color attribute for the edge from `u` to `v` with key `k` while `for (u, v, k, c) in G.edges(data='color', keys=True, default="red"):` iterates through all the edges yielding the color attribute with default `'red'` if no color attribute exists.

Edges are returned as tuples with optional data and keys in the order (node, neighbor, key, data). If `keys=True` is not provided, the tuples will just be (node, neighbor, data), but multiple tuples with the same node and neighbor will be generated when multiple edges exist between two nodes.

Parameters

nbunch
[single node, container, or all nodes (default= all nodes)] The view will only report edges from 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 (u, v, k) tuples or (u, v, k, d) tuples if data is also requested.

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

edges
[MultiEdgeView] A view of edge attributes, usually it iterates over (u, v) (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']`.

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.MultiGraph()
>>> nx.add_path(G, [0, 1, 2])
>>> key = G.add_edge(2, 3, weight=5)
>>> key2 = G.add_edge(2, 1, weight=2)  # multi-edge
>>> [e for e in G.edges()]
[(0, 1), (1, 2), (1, 2), (2, 3)]
>>> G.edges.data()  # default data is {} (empty dict)
MultiEdgeDataView([(0, 1, {}), (1, 2, {}), (1, 2, {'weight': 2}), (2, 3, {'weight
↪': 5})])
>>> G.edges.data("weight", default=1)
MultiEdgeDataView([(0, 1, 1), (1, 2, 1), (1, 2, 2), (2, 3, 5)])
>>> G.edges(keys=True)  # default keys are integers
MultiEdgeView([(0, 1, 0), (1, 2, 0), (1, 2, 1), (2, 3, 0)])
>>> G.edges.data(keys=True)
MultiEdgeDataView([(0, 1, 0, {}), (1, 2, 0, {}), (1, 2, 1, {'weight': 2}), (2, 3,␣
↪0, {'weight': 5})])
>>> G.edges.data("weight", default=1, keys=True)
MultiEdgeDataView([(0, 1, 0, 1), (1, 2, 0, 1), (1, 2, 1, 2), (2, 3, 0, 5)])
>>> G.edges([0, 3])  # Note ordering of tuples from listed sources
MultiEdgeDataView([(0, 1), (3, 2)])
>>> G.edges([0, 3, 2, 1])  # Note ordering of tuples
MultiEdgeDataView([(0, 1), (3, 2), (2, 1), (2, 1)])
>>> G.edges(0)
MultiEdgeDataView([(0, 1)])
```

MultiGraph.has_edge

`MultiGraph.`**`has_edge`**(*u*, *v*, *key=None*)

Returns True if the graph has an edge between nodes u and v.

This is the same as `v in G[u] or key in G[u][v]` without KeyError exceptions.

Parameters

u, v
: [nodes] Nodes can be, for example, strings or numbers.

key
: [hashable identifier, optional (default=None)] If specified return True only if the edge with key is found.

Returns

edge_ind
: [bool] True if edge is in the graph, False otherwise.

Examples

Can be called either using two nodes u, v, an edge tuple (u, v), or an edge tuple (u, v, key).

```
>>> G = nx.MultiGraph()  # or MultiDiGraph
>>> nx.add_path(G, [0, 1, 2, 3])
>>> G.has_edge(0, 1)  # using two nodes
True
>>> e = (0, 1)
>>> G.has_edge(*e)  #  e is a 2-tuple (u, v)
True
>>> G.add_edge(0, 1, key="a")
'a'
>>> G.has_edge(0, 1, key="a")  # specify key
True
>>> G.has_edge(1, 0, key="a")  # edges aren't directed
True
>>> e = (0, 1, "a")
>>> G.has_edge(*e)  # e is a 3-tuple (u, v, 'a')
True
```

The following syntax are equivalent:

```
>>> G.has_edge(0, 1)
True
>>> 1 in G[0]  # though this gives :exc:`KeyError` if 0 not in G
True
>>> 0 in G[1]  # other order; also gives :exc:`KeyError` if 0 not in G
True
```

MultiGraph.get_edge_data

`MultiGraph.`**`get_edge_data`**(*u*, *v*, *key=None*, *default=None*)

Returns the attribute dictionary associated with edge (u, v, key).

If a key is not provided, returns a dictionary mapping edge keys to attribute dictionaries for each edge between u and v.

This is identical to `G[u][v][key]` except the default is returned instead of an exception is the edge doesn't exist.

Parameters

u, v
[nodes]

default
[any Python object (default=None)] Value to return if the specific edge (u, v, key) is not found, OR if there are no edges between u and v and no key is specified.

key
[hashable identifier, optional (default=None)] Return data only for the edge with specified key, as an attribute dictionary (rather than a dictionary mapping keys to attribute dictionaries).

Returns

edge_dict
[dictionary] The edge attribute dictionary, OR a dictionary mapping edge keys to attribute dictionaries for each of those edges if no specific key is provided (even if there's only one edge between u and v).

Examples

```
>>> G = nx.MultiGraph()  # or MultiDiGraph
>>> key = G.add_edge(0, 1, key="a", weight=7)
>>> G[0][1]["a"]  # key='a'
{'weight': 7}
>>> G.edges[0, 1, "a"]  # key='a'
{'weight': 7}
```

Warning: we protect the graph data structure by making `G.edges` and `G[1][2]` read-only dict-like structures. However, you can assign values to attributes in e.g. `G.edges[1, 2, 'a']` or `G[1][2]['a']` using an additional bracket as shown next. You need to specify all edge info to assign to the edge data associated with an edge.

```
>>> G[0][1]["a"]["weight"] = 10
>>> G.edges[0, 1, "a"]["weight"] = 10
>>> G[0][1]["a"]["weight"]
10
>>> G.edges[1, 0, "a"]["weight"]
10
```

```
>>> G = nx.MultiGraph()  # or MultiDiGraph
>>> nx.add_path(G, [0, 1, 2, 3])
>>> G.edges[0, 1, 0]["weight"] = 5
>>> G.get_edge_data(0, 1)
{0: {'weight': 5}}
>>> e = (0, 1)
>>> G.get_edge_data(*e)  # tuple form
{0: {'weight': 5}}
>>> G.get_edge_data(3, 0)  # edge not in graph, returns None
>>> G.get_edge_data(3, 0, default=0)  # edge not in graph, return default
0
>>> G.get_edge_data(1, 0, 0)  # specific key gives back
{'weight': 5}
```

MultiGraph.neighbors

`MultiGraph.`**`neighbors`**`(`*`n`*`)`

Returns an iterator over all neighbors of node n.

This is identical to `iter(G[n])`

Parameters

n
[node] A node in the graph

Returns

neighbors
[iterator] An iterator over all neighbors of node n

Raises

NetworkXError
If the node n is not in the graph.

Notes

Alternate ways to access the neighbors are `G.adj[n]` or `G[n]`:

```
>>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge("a", "b", weight=7)
>>> G["a"]
AtlasView({'b': {'weight': 7}})
>>> G = nx.path_graph(4)
>>> [n for n in G[0]]
[1]
```

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> [n for n in G.neighbors(0)]
[1]
```

MultiGraph.adj

property MultiGraph.**adj**

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 the color of the edge `(3, 2, 0)` to `"blue"`.

Iterating over G.adj behaves like a dict. Useful idioms include `for nbr, edgesdict in G.adj[n].items()`:.

The neighbor information is also provided by subscripting the graph.

Examples

```
>>> e = [(1, 2), (1, 2), (1, 3), (3, 4)]  # list of edges
>>> G = nx.MultiGraph(e)
>>> G.edges[1, 2, 0]["weight"] = 3
>>> result = set()
>>> for edgekey, data in G[1][2].items():
...     result.add(data.get('weight', 1))
>>> result
{1, 3}
```

For directed graphs, `G.adj` holds outgoing (successor) info.

MultiGraph.__getitem__

MultiGraph.**__getitem__**(*n*)

Returns a dict of neighbors of node n. Use: 'G[n]'.

Parameters

n
[node] A node in the graph.

Returns

adj_dict
[dictionary] The adjacency dictionary for nodes connected to n.

Notes

G[n] is the same as G.adj[n] and similar to G.neighbors(n) (which is an iterator over G.adj[n])

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G[0]
AtlasView({1: {}})
```

MultiGraph.adjacency

MultiGraph.**adjacency**()

Returns an iterator over (node, adjacency dict) tuples for all nodes.

For directed graphs, only outgoing neighbors/adjacencies are included.

Returns

adj_iter
[iterator] An iterator over (node, adjacency dictionary) for all nodes in the graph.

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> [(n, nbrdict) for n, nbrdict in G.adjacency()]
[(0, {1: {}}), (1, {0: {}, 2: {}}), (2, {1: {}, 3: {}}), (3, {2: {}})]
```

MultiGraph.nbunch_iter

MultiGraph.**nbunch_iter**(*nbunch=None*)

Returns an iterator over nodes contained in nbunch that are also in the graph.

The nodes in nbunch are checked for membership in the graph and if not are silently ignored.

Parameters

nbunch
[single node, container, or all nodes (default= all nodes)] The view will only report edges incident to these nodes.

Returns

niter
[iterator] An iterator over nodes in nbunch that are also in the graph. If nbunch is None, iterate over all nodes in the graph.

Raises

NetworkXError
If nbunch is not a node or sequence of nodes. If a node in nbunch is not hashable.

See also:

Graph.__iter__

Notes

When nbunch is an iterator, the returned iterator yields values directly from nbunch, becoming exhausted when nbunch is exhausted.

To test whether nbunch is a single node, one can use "if nbunch in self:", even after processing with this routine.

If nbunch is not a node or a (possibly empty) sequence/iterator or None, a *NetworkXError* is raised. Also, if any object in nbunch is not hashable, a *NetworkXError* is raised.

Counting nodes edges and neighbors

MultiGraph.order()	Returns the number of nodes in the graph.
MultiGraph.number_of_nodes()	Returns the number of nodes in the graph.
MultiGraph.__len__()	Returns the number of nodes in the graph.
MultiGraph.degree	A DegreeView for the Graph as G.degree or G.degree().
MultiGraph.size([weight])	Returns the number of edges or total of all edge weights.
MultiGraph.number_of_edges([u, v])	Returns the number of edges between two nodes.

MultiGraph.order

MultiGraph.**order**()

Returns the number of nodes in the graph.

Returns

nnodes
[int] The number of nodes in the graph.

See also:

number_of_nodes
identical method

__len__
identical method

Examples

```
>>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.order()
3
```

MultiGraph.number_of_nodes

MultiGraph.**number_of_nodes**()

Returns the number of nodes in the graph.

Returns

nnodes
[int] The number of nodes in the graph.

See also:

order
identical method

__len__
identical method

Examples

```
>>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.number_of_nodes()
3
```

MultiGraph.__len__

MultiGraph.**__len__**()

Returns the number of nodes in the graph. Use: 'len(G)'.

Returns

nnodes
[int] The number of nodes in the graph.

See also:

number_of_nodes
identical method

order
identical method

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> len(G)
4
```

MultiGraph.degree

property MultiGraph.**degree**

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

MultiDegreeView or int
If multiple nodes are requested (the default), returns a `MultiDegreeView` mapping nodes to their degree. If a single node is requested, returns the degree of the node as an integer.

Examples

```
>>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> nx.add_path(G, [0, 1, 2, 3])
>>> G.degree(0)  # node 0 with degree 1
1
>>> list(G.degree([0, 1]))
[(0, 1), (1, 2)]
```

MultiGraph.size

MultiGraph.**size**(*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
```

MultiGraph.number_of_edges

MultiGraph.**number_of_edges**(*u=None*, *v=None*)

Returns the number of edges between two nodes.

Parameters

u, v
[nodes, optional (Gefault=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 multigraphs, this method counts the total number of edges in the graph:

```
>>> G = nx.MultiGraph()
>>> G.add_edges_from([(0, 1), (0, 1), (1, 2)])
[0, 1, 0]
>>> 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)
2
```

For directed multigraphs, this method can count the total number of directed edges from u to v:

```
>>> G = nx.MultiDiGraph()
>>> G.add_edges_from([(0, 1), (0, 1), (1, 0)])
[0, 1, 0]
>>> G.number_of_edges(0, 1)
2
>>> G.number_of_edges(1, 0)
1
```

Making copies and subgraphs

MultiGraph.copy([as_view])	Returns a copy of the graph.
MultiGraph.to_undirected([as_view])	Returns an undirected copy of the graph.
MultiGraph.to_directed([as_view])	Returns a directed representation of the graph.
MultiGraph.subgraph(nodes)	Returns a SubGraph view of the subgraph induced on `nodes`.
MultiGraph.edge_subgraph(edges)	Returns the subgraph induced by the specified edges.

MultiGraph.copy

`MultiGraph.`**`copy`**(*as_view=False*)

Returns a copy of the graph.

The copy method by default returns an independent shallow copy of the graph and attributes. That is, if an attribute is a container, that container is shared by the original an the copy. Use Python's `copy.deepcopy` for new containers.

If `as_view` is True then a view is returned instead of a copy.

Parameters

as_view
[bool, optional (default=False)] If True, the returned graph-view provides a read-only view of the original graph without actually copying any data.

Returns

G
[Graph] A copy of the graph.

See also:

to_directed
return a directed copy of the graph.

Notes

All copies reproduce the graph structure, but data attributes may be handled in different ways. There are four types of copies of a graph that people might want.

Deepcopy – A "deepcopy" copies the graph structure as well as all data attributes and any objects they might contain. The entire graph object is new so that changes in the copy do not affect the original object. (see Python's copy.deepcopy)

Data Reference (Shallow) – For a shallow copy the graph structure is copied but the edge, node and graph attribute dicts are references to those in the original graph. This saves time and memory but could cause confusion if you change an attribute in one graph and it changes the attribute in the other. NetworkX does not provide this level of shallow copy.

Independent Shallow – This copy creates new independent attribute dicts and then does a shallow copy of the attributes. That is, any attributes that are containers are shared between the new graph and the original. This is exactly what `dict.copy()` provides. You can obtain this style copy using:

```
>>> G = nx.path_graph(5)
>>> H = G.copy()
>>> H = G.copy(as_view=False)
>>> H = nx.Graph(G)
>>> H = G.__class__(G)
```

Fresh Data – For fresh data, the graph structure is copied while new empty data attribute dicts are created. The resulting graph is independent of the original and it has no edge, node or graph attributes. Fresh copies are not enabled. Instead use:

```
>>> H = G.__class__()
>>> H.add_nodes_from(G)
>>> H.add_edges_from(G.edges)
```

View – Inspired by dict-views, graph-views act like read-only versions of the original graph, providing a copy of the original structure without requiring any memory for copying the information.

See the Python copy module for more information on shallow and deep copies, https://docs.python.org/3/library/copy.html.

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> H = G.copy()
```

MultiGraph.to_undirected

MultiGraph.**to_undirected**(*as_view=False*)

Returns an undirected copy of the graph.

Returns

G

[Graph/MultiGraph] A deepcopy of the graph.

See also:

copy, ***add_edge***, ***add_edges_from***

Notes

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 = nx.MultiGraph(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 MultiGraph to use dict-like objects in the data structure, those changes do not transfer to the MultiGraph created by this method.

Examples

```
>>> G = nx.MultiGraph([(0, 1), (0, 1), (1, 2)])
>>> H = G.to_directed()
>>> list(H.edges)
[(0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 2, 0), (2, 1, 0)]
>>> G2 = H.to_undirected()
>>> list(G2.edges)
[(0, 1, 0), (0, 1, 1), (1, 2, 0)]
```

MultiGraph.to_directed

MultiGraph.**to_directed**(*as_view=False*)

Returns a directed representation of the graph.

Returns

G

[MultiDiGraph] A directed graph with the same name, same nodes, and with each edge (u, v, k, data) replaced by two directed edges (u, v, k, data) and (v, u, k, data).

Notes

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 D=MultiDiGraph(G) 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 MultiGraph to use dict-like objects in the data structure, those changes do not transfer to the MultiDiGraph created by this method.

Examples

```
>>> G = nx.MultiGraph()
>>> G.add_edge(0, 1)
0
>>> G.add_edge(0, 1)
1
>>> H = G.to_directed()
>>> list(H.edges)
[(0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1)]
```

If already directed, return a (deep) copy

```
>>> G = nx.MultiDiGraph()
>>> G.add_edge(0, 1)
0
>>> H = G.to_directed()
>>> list(H.edges)
[(0, 1, 0)]
```

MultiGraph.subgraph

MultiGraph.**subgraph**(*nodes*)

Returns a SubGraph view of the subgraph induced on *nodes*.

The induced subgraph of the graph contains the nodes in *nodes* and the edges between those nodes.

Parameters

nodes
[list, iterable] A container of nodes which will be iterated through once.

Returns

G
[SubGraph View] A subgraph view of the graph. The graph structure cannot be changed but node/edge attributes can and are shared with the original graph.

Notes

The graph, edge and node attributes are shared with the original graph. Changes to the graph structure is ruled out by the view, but changes to attributes are reflected in the original graph.

To create a subgraph with its own copy of the edge/node attributes use: G.subgraph(nodes).copy()

For an inplace reduction of a graph to a subgraph you can remove nodes: G.remove_nodes_from([n for n in G if n not in set(nodes)])

Subgraph views are sometimes NOT what you want. In most cases where you want to do more than simply look at the induced edges, it makes more sense to just create the subgraph as its own graph with code like:

```
# Create a subgraph SG based on a (possibly multigraph) G
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)]
```

MultiGraph.edge_subgraph

`MultiGraph.`**`edge_subgraph`**(*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)]
```

2.2.4 MultiDiGraph—Directed graphs with self loops and parallel edges

Overview

`class MultiDiGraph`(*incoming_graph_data=None*, *multigraph_input=None*, ***attr*)

A directed graph class that can store multiedges.

Multiedges are multiple edges between two nodes. Each edge can hold optional data or attributes.

A MultiDiGraph holds directed edges. Self loops are allowed.

Nodes can be arbitrary (hashable) Python objects with optional key/value attributes. By convention `None` is not used as a node.

Edges are represented as links between nodes with optional key/value 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 any format that is supported by the to_networkx_graph() function, currently including edge list, dict of dicts, dict of lists, NetworkX graph, 2D NumPy array, SciPy sparse matrix, or 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, `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:

Graph
DiGraph
MultiGraph

Examples

Create an empty graph structure (a "null graph") with no nodes and no edges.

```
>>> G = nx.MultiDiGraph()
```

G can be grown in several ways.

Nodes:

Add one node at a time:

```
>>> G.add_node(1)
```

Add the nodes from any container (a list, dict, set or even the lines from a file or the nodes from another graph).

```
>>> G.add_nodes_from([2, 3])
>>> G.add_nodes_from(range(100, 110))
>>> H = nx.path_graph(10)
>>> G.add_nodes_from(H)
```

In addition to strings and integers any hashable Python object (except None) can represent a node, e.g. a customized node object, or even another Graph.

```
>>> G.add_node(H)
```

Edges:

G can also be grown by adding edges.

Add one edge,

```
>>> key = G.add_edge(1, 2)
```

a list of edges,

```
>>> keys = G.add_edges_from([(1, 2), (1, 3)])
```

or a collection of edges,

```
>>> keys = G.add_edges_from(H.edges)
```

If some edges connect nodes not yet in the graph, the nodes are added automatically. If an edge already exists, an additional edge is created and stored using a key to identify the edge. By default the key is the lowest unused integer.

```
>>> keys = G.add_edges_from([(4, 5, dict(route=282)), (4, 5, dict(route=37))])
>>> G[4]
AdjacencyView({5: {0: {}, 1: {'route': 282}, 2: {'route': 37}}})
```

Attributes:

Each graph, node, and edge can hold key/value attribute pairs in an associated attribute dictionary (the keys must be hashable). By default these are empty, but can be added or changed using add_edge, add_node or direct manipulation of the attribute dictionaries named graph, node and edge respectively.

```
>>> G = nx.MultiDiGraph(day="Friday")
>>> G.graph
{'day': 'Friday'}
```

Add node attributes using add_node(), add_nodes_from() or G.nodes

```
>>> G.add_node(1, time="5pm")
>>> G.add_nodes_from([3], time="2pm")
>>> G.nodes[1]
{'time': '5pm'}
>>> G.nodes[1]["room"] = 714
>>> del G.nodes[1]["room"]  # remove attribute
>>> list(G.nodes(data=True))
[(1, {'time': '5pm'}), (3, {'time': '2pm'})]
```

Add edge attributes using add_edge(), add_edges_from(), subscript notation, or G.edges.

```
>>> key = G.add_edge(1, 2, weight=4.7)
>>> keys = G.add_edges_from([(3, 4), (4, 5)], color="red")
>>> keys = G.add_edges_from([(1, 2, {"color": "blue"}), (2, 3, {"weight": 8})])
>>> G[1][2][0]["weight"] = 4.7
>>> G.edges[1, 2, 0]["weight"] = 4
```

Warning: we protect the graph data structure by making `G.edges[1, 2, 0]` a read-only dict-like structure. However, you can assign to attributes in e.g. `G.edges[1, 2, 0]`. Thus, use 2 sets of brackets to add/change data attributes: `G.edges[1, 2, 0]['weight'] = 4` (for multigraphs the edge key is required: `MG.edges[u, v, key][name] = value`).

Shortcuts:

Many common graph features allow python syntax to speed reporting.

```
>>> 1 in G  # check if node in graph
True
>>> [n for n in G if n < 3]  # iterate through nodes
[1, 2]
>>> len(G)  # number of nodes in graph
5
>>> G[1]  # adjacency dict-like view mapping neighbor -> edge key -> edge␣
↪attributes
AdjacencyView({2: {0: {'weight': 4}, 1: {'color': 'blue'}}})
```

Often the best way to traverse all edges of a graph is via the neighbors. The neighbors are available as an adjacency-view `G.adj` object or via the method `G.adjacency()`.

```
>>> for n, nbrsdict in G.adjacency():
...     for nbr, keydict in nbrsdict.items():
...         for key, eattr in keydict.items():
...             if "weight" in eattr:
...                 # Do something useful with the edges
...                 pass
```

But the edges() method is often more convenient:

```
>>> for u, v, keys, weight in G.edges(data="weight", keys=True):
...     if weight is not None:
...         # Do something useful with the edges
...         pass
```

Reporting:

Simple graph information is obtained using methods and object-attributes. Reporting usually provides views instead of containers to reduce memory usage. The views update as the graph is updated similarly to dict-views. The objects *nodes*, *edges* and *adj* provide access to data attributes via lookup (e.g. `nodes[n]`, `edges[u, v, k]`, `adj[u][v]`) and iteration (e.g. `nodes.items()`, `nodes.data('color')`, `nodes.data('color', default='blue')` and similarly for *edges*) Views exist for *nodes*, *edges*, *neighbors()*/*adj* and *degree*.

For details on these and other miscellaneous methods, see below.

Subclasses (Advanced):

The MultiDiGraph class uses a dict-of-dict-of-dict-of-dict structure. The outer dict (node_dict) holds adjacency information keyed by node. The next dict (adjlist_dict) represents the adjacency information and holds edge_key dicts keyed by neighbor. The edge_key dict holds each edge_attr dict keyed by edge key. The inner dict (edge_attr_dict) represents the edge data and holds edge attribute values keyed by attribute names.

Each of these four dicts in the dict-of-dict-of-dict-of-dict structure can be replaced 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. 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
```

Methods

Adding and Removing Nodes and Edges

MultiDiGraph.__init__([incoming_graph_data, ...])	Initialize a graph with edges, name, or graph attributes.
MultiDiGraph.add_node(node_for_adding, **attr)	Add a single node `node_for_adding` and update node attributes.
MultiDiGraph.add_nodes_from(...)	Add multiple nodes.
MultiDiGraph.remove_node(n)	Remove node n.
MultiDiGraph.remove_nodes_from(nodes)	Remove multiple nodes.
MultiDiGraph.add_edge(u_for_edge, v_for_edge)	Add an edge between u and v.
MultiDiGraph.add_edges_from(ebunch_to_add, ...)	Add all the edges in ebunch_to_add.
MultiDiGraph.add_weighted_edges_from(...[, ...])	Add weighted edges in `ebunch_to_add` with specified weight attr
MultiDiGraph.new_edge_key(u, v)	Returns an unused key for edges between nodes `u` and `v`.
MultiDiGraph.remove_edge(u, v[, key])	Remove an edge between u and v.
MultiDiGraph.remove_edges_from(ebunch)	Remove all edges specified in ebunch.
MultiDiGraph.update([edges, nodes])	Update the graph using nodes/edges/graphs as input.
MultiDiGraph.clear()	Remove all nodes and edges from the graph.
MultiDiGraph.clear_edges()	Remove all edges from the graph without altering nodes.

MultiDiGraph.__init__

MultiDiGraph.**__init__**(*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, `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'}
```

MultiDiGraph.add_node

MultiDiGraph.**add_node**(*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

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.

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))
```

MultiDiGraph.add_nodes_from

`MultiDiGraph.`**`add_nodes_from`**(*nodes_for_adding*, ***attr*)

Add multiple nodes.

Parameters

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.DiGraph([(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))
```

MultiDiGraph.remove_node

MultiDiGraph.**remove_node**(*n*)

Remove node n.

Removes the node n and all adjacent edges. Attempting to remove a non-existent node will raise an exception.

Parameters

n
[node] A node in the graph

Raises

NetworkXError
If n is not in the graph.

See also:

remove_nodes_from

Examples

```
>>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> list(G.edges)
[(0, 1), (1, 2)]
>>> G.remove_node(1)
>>> list(G.edges)
[]
```

MultiDiGraph.remove_nodes_from

MultiDiGraph.**remove_nodes_from**(*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`.

Examples

```
>>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> e = list(G.nodes)
>>> e
[0, 1, 2]
>>> G.remove_nodes_from(e)
>>> list(G.nodes)
[]
```

Evaluate an iterator over a graph if using it to modify the same graph

```
>>> G = nx.DiGraph([(0, 1), (1, 2), (3, 4)])
>>> # this command will fail, as the graph's dict is modified during iteration
>>> # G.remove_nodes_from(n for n in G.nodes if n < 2)
>>> # this command will work, since the dictionary underlying graph is not␣
↪modified
>>> G.remove_nodes_from(list(n for n in G.nodes if n < 2))
```

MultiDiGraph.add_edge

`MultiDiGraph.`**`add_edge`**(*u_for_edge*, *v_for_edge*, *key=None*, ***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_for_edge, v_for_edge
: [nodes] Nodes can be, for example, strings or numbers. 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})
```

MultiDiGraph.add_edges_from

MultiDiGraph.**add_edges_from**(*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 can be:

- 2-tuples (u, v) or
- 3-tuples (u, v, d) for an edge data dict d, or
- 3-tuples (u, v, k) for not iterable key k, or
- 4-tuples (u, v, k, d) for an edge with data and key k

attr
: [keyword arguments, optional] Edge data (or labels or objects) can be assigned using keyword arguments.

Returns

A list of edge keys assigned to the edges in `ebunch`.

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))
```

MultiDiGraph.add_weighted_edges_from

MultiDiGraph.**add_weighted_edges_from**(*ebunch_to_add*, *weight='weight'*, ***attr*)

Add weighted edges in `ebunch_to_add` with specified weight attr

Parameters

ebunch_to_add
[container of edges] Each edge given in the list or container will be added to the graph. The edges must be given as 3-tuples (u, v, w) where w is a number.

weight
[string, optional (default= 'weight')] The attribute name for the edge weights to be added.

attr
[keyword arguments, optional (default= no attributes)] Edge attributes to add/update for all edges.

See also:

add_edge
add a single edge

add_edges_from
add multiple edges

Notes

Adding the same edge twice for Graph/DiGraph simply updates the edge data. For MultiGraph/MultiDiGraph, duplicate edges are stored.

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_weighted_edges_from`.

Examples

```
>>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_weighted_edges_from([(0, 1, 3.0), (1, 2, 7.5)])
```

Evaluate an iterator over edges before passing it

```
>>> G = nx.Graph([(1, 2), (2, 3), (3, 4)])
>>> weight = 0.1
>>> # Grow graph by one new node, adding edges to all existing nodes.
>>> # wrong way - will raise RuntimeError
>>> # G.add_weighted_edges_from(((5, n, weight) for n in G.nodes))
>>> # correct way - note that there will be no self-edge for node 5
>>> G.add_weighted_edges_from(list((5, n, weight) for n in G.nodes))
```

MultiDiGraph.new_edge_key

MultiDiGraph.**new_edge_key**(*u*, *v*)

Returns an unused key for edges between nodes u and v.

The nodes u and v do not need to be already in the graph.

Parameters

u, v
[nodes]

Returns

key
[int]

Notes

In the standard MultiGraph class the new key is the number of existing edges between u and v (increased if necessary to ensure unused). The first edge will have key 0, then 1, etc. If an edge is removed further new_edge_keys may not be in this order.

MultiDiGraph.remove_edge

MultiDiGraph.**remove_edge**(*u*, *v*, *key=None*)

Remove an edge between u and v.

Parameters

u, v
[nodes] Remove an edge between nodes u and v.

key
[hashable identifier, optional (default=None)] Used to distinguish multiple edges between a pair of nodes. If None, remove a single edge between u and v. If there are multiple edges, removes the last edge added in terms of insertion order.

Raises

NetworkXError
If there is not an edge between u and v, or if there is no edge with the specified key.

See also:

remove_edges_from
remove a collection of edges

Examples

```
>>> G = nx.MultiDiGraph()
>>> nx.add_path(G, [0, 1, 2, 3])
>>> G.remove_edge(0, 1)
>>> e = (1, 2)
>>> G.remove_edge(*e)  # unpacks e from an edge tuple
```

For multiple edges

```
>>> G = nx.MultiDiGraph()
>>> G.add_edges_from([(1, 2), (1, 2), (1, 2)])  # key_list returned
[0, 1, 2]
```

When `key=None` (the default), edges are removed in the opposite order that they were added:

```
>>> G.remove_edge(1, 2)
>>> G.edges(keys=True)
OutMultiEdgeView([(1, 2, 0), (1, 2, 1)])
```

For edges with keys

```
>>> G = nx.MultiDiGraph()
>>> G.add_edge(1, 2, key="first")
'first'
>>> G.add_edge(1, 2, key="second")
'second'
>>> G.remove_edge(1, 2, key="first")
>>> G.edges(keys=True)
OutMultiEdgeView([(1, 2, 'second')])
```

MultiDiGraph.remove_edges_from

MultiDiGraph.**remove_edges_from**(*ebunch*)

Remove all edges specified in ebunch.

Parameters

ebunch: list or container of edge tuples

Each edge given in the list or container will be removed from the graph. The edges can be:

- 2-tuples (u, v) A single edge between u and v is removed.
- 3-tuples (u, v, key) The edge identified by key is removed.
- 4-tuples (u, v, key, data) where data 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)
```

Removing multiple copies of edges

```
>>> G = nx.MultiGraph()
>>> keys = G.add_edges_from([(1, 2), (1, 2), (1, 2)])
>>> G.remove_edges_from([(1, 2), (2, 1)])  # edges aren't directed
>>> list(G.edges())
[(1, 2)]
>>> G.remove_edges_from([(1, 2), (1, 2)])  # silently ignore extra copy
>>> list(G.edges)  # now empty graph
[]
```

When the edge is a 2-tuple `(u, v)` but there are multiple edges between u and v in the graph, the most recent edge (in terms of insertion order) is removed.

```
>>> G = nx.MultiGraph()
>>> for key in ("x", "y", "a"):
...     k = G.add_edge(0, 1, key=key)
>>> G.edges(keys=True)
MultiEdgeView([(0, 1, 'x'), (0, 1, 'y'), (0, 1, 'a')])
>>> G.remove_edges_from([(0, 1)])
>>> G.edges(keys=True)
MultiEdgeView([(0, 1, 'x'), (0, 1, 'y')])
```

MultiDiGraph.update

`MultiDiGraph.`**`update`**(*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.

See also:

add_edges_from
add multiple edges to a graph

add_nodes_from
add multiple nodes to a graph

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}}
>>> e = [
...     (u, v, {"weight": d})
...     for u, nbrs in adj.items()
...     for v, d in nbrs.items()
... ]
>>> DG.update(edges=e, nodes=adj)
```

```
>>> # dict-of-dict-of-dict
>>> adj = {1: {2: {"weight": 1.3}, 3: {"color": 0.7, "weight": 1.2}}}
>>> e = [
...     (u, v, {"weight": d})
...     for u, nbrs in adj.items()
...     for v, d in nbrs.items()
... ]
>>> DG.update(edges=e, nodes=adj)
```

```
>>> # predecessor adjacency (dict-of-set)
>>> pred = {1: {2, 3}, 2: {3}, 3: {3}}
>>> e = [(v, u) for u, nbrs in pred.items() for v in nbrs]
```

```
>>> # MultiGraph dict-of-dict-of-dict-of-attribute
>>> MDG = nx.MultiDiGraph()
>>> adj = {
...     1: {2: {0: {"weight": 1.3}, 1: {"weight": 1.2}}},
...     3: {2: {0: {"weight": 0.7}}},
... }
>>> e = [
...     (u, v, ekey, d)
```

(continues on next page)

(continued from previous page)

```
...     for u, nbrs in adj.items()
...     for v, keydict in nbrs.items()
...     for ekey, d in keydict.items()
... ]
>>> MDG.update(edges=e)
```

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)
```

MultiDiGraph.clear

MultiDiGraph.**clear**()

Remove all nodes and edges from the graph.

This also removes the name, and all graph, node, and edge attributes.

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.clear()
>>> list(G.nodes)
[]
>>> list(G.edges)
[]
```

MultiDiGraph.clear_edges

MultiDiGraph.**clear_edges**()

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)
[]
```

Reporting nodes edges and neighbors

MultiDiGraph.nodes	A NodeView of the Graph as G.nodes or G.nodes().
MultiDiGraph.__iter__()	Iterate over the nodes.
MultiDiGraph.has_node(n)	Returns True if the graph contains the node n.
MultiDiGraph.__contains__(n)	Returns True if n is a node, False otherwise.
MultiDiGraph.edges	An OutMultiEdgeView of the Graph as G.edges or G.edges().
MultiDiGraph.out_edges	An OutMultiEdgeView of the Graph as G.edges or G.edges().
MultiDiGraph.in_edges	A view of the in edges of the graph as G.in_edges or G.in_edges().
MultiDiGraph.has_edge(u, v[, key])	Returns True if the graph has an edge between nodes u and v.
MultiDiGraph.get_edge_data(u, v[, key, default])	Returns the attribute dictionary associated with edge (u, v, key).
MultiDiGraph.neighbors(n)	Returns an iterator over successor nodes of n.
MultiDiGraph.adj	Graph adjacency object holding the neighbors of each node.
MultiDiGraph.__getitem__(n)	Returns a dict of neighbors of node n.
MultiDiGraph.successors(n)	Returns an iterator over successor nodes of n.
MultiDiGraph.succ	Graph adjacency object holding the successors of each node.
MultiDiGraph.predecessors(n)	Returns an iterator over predecessor nodes of n.
MultiDiGraph.succ	Graph adjacency object holding the successors of each node.
MultiDiGraph.adjacency()	Returns an iterator over (node, adjacency dict) tuples for all nodes.
MultiDiGraph.nbunch_iter([nbunch])	Returns an iterator over nodes contained in nbunch that are also in the graph.

MultiDiGraph.nodes

property MultiDiGraph.**nodes**

A NodeView of the Graph as G.nodes or G.nodes().

Can be used as `G.nodes` for data lookup and for set-like operations. Can also be used as `G.nodes(data='color', default=None)` to return a NodeDataView which reports specific node data but no set operations. It presents a dict-like interface as well with `G.nodes.items()` iterating over `(node, nodedata)` 2-tuples and `G.nodes[3]['foo']` providing the value of the `foo` attribute for node `3`. In

addition, a view `G.nodes.data('foo')` provides a dict-like interface to the `foo` attribute of each node. `G.nodes.data('foo', default=1)` provides a default for nodes that do not have attribute `foo`.

Parameters

data
: [string or bool, optional (default=False)] The node attribute returned in 2-tuple (n, ddict[data]). If True, return entire node attribute dict as (n, ddict). If False, return just the nodes n.

default
: [value, optional (default=None)] Value used for nodes that don't have the requested attribute. Only relevant if data is not True or False.

Returns

NodeView
: Allows set-like operations over the nodes as well as node attribute dict lookup and calling to get a NodeDataView. A NodeDataView iterates over `(n, data)` and has no set operations. A NodeView iterates over `n` and includes set operations.

  When called, if data is False, an iterator over nodes. Otherwise an iterator of 2-tuples (node, attribute value) where the attribute is specified in `data`. If data is True then the attribute becomes the entire data dictionary.

Notes

If your node data is not needed, it is simpler and equivalent to use the expression `for n in G`, or `list(G)`.

Examples

There are two simple ways of getting a list of all nodes in the graph:

```
>>> G = nx.path_graph(3)
>>> list(G.nodes)
[0, 1, 2]
>>> list(G)
[0, 1, 2]
```

To get the node data along with the nodes:

```
>>> G.add_node(1, time="5pm")
>>> G.nodes[0]["foo"] = "bar"
>>> list(G.nodes(data=True))
[(0, {'foo': 'bar'}), (1, {'time': '5pm'}), (2, {})]
>>> list(G.nodes.data())
[(0, {'foo': 'bar'}), (1, {'time': '5pm'}), (2, {})]
```

```
>>> list(G.nodes(data="foo"))
[(0, 'bar'), (1, None), (2, None)]
>>> list(G.nodes.data("foo"))
[(0, 'bar'), (1, None), (2, None)]
```

```
>>> list(G.nodes(data="time"))
[(0, None), (1, '5pm'), (2, None)]
>>> list(G.nodes.data("time"))
[(0, None), (1, '5pm'), (2, None)]
```

```
>>> list(G.nodes(data="time", default="Not Available"))
[(0, 'Not Available'), (1, '5pm'), (2, 'Not Available')]
>>> list(G.nodes.data("time", default="Not Available"))
[(0, 'Not Available'), (1, '5pm'), (2, 'Not Available')]
```

If some of your nodes have an attribute and the rest are assumed to have a default attribute value you can create a dictionary from node/attribute pairs using the `default` keyword argument to guarantee the value is never None:

```
>>> G = nx.Graph()
>>> G.add_node(0)
>>> G.add_node(1, weight=2)
>>> G.add_node(2, weight=3)
>>> dict(G.nodes(data="weight", default=1))
{0: 1, 1: 2, 2: 3}
```

MultiDiGraph.__iter__

`MultiDiGraph.__iter__()`

Iterate over the nodes. Use: 'for n in G'.

Returns

niter
[iterator] An iterator over all nodes in the graph.

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> [n for n in G]
[0, 1, 2, 3]
>>> list(G)
[0, 1, 2, 3]
```

MultiDiGraph.has_node

`MultiDiGraph.has_node(n)`

Returns True if the graph contains the node n.

Identical to `n in G`

Parameters

n
[node]

Examples

```
>>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.has_node(0)
True
```

It is more readable and simpler to use

```
>>> 0 in G
True
```

MultiDiGraph.__contains__

MultiDiGraph.**__contains__**(*n*)

Returns True if n is a node, False otherwise. Use: 'n in G'.

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> 1 in G
True
```

MultiDiGraph.edges

property MultiDiGraph.**edges**

An OutMultiEdgeView of the Graph as G.edges or G.edges().

edges(self, nbunch=None, data=False, keys=False, default=None)

The OutMultiEdgeView provides set-like operations on the edge-tuples as well as edge attribute lookup. When called, it also provides an EdgeDataView object which allows control of access to edge attributes (but does not provide set-like operations). Hence, `G.edges[u, v, k]['color']` provides the value of the color attribute for the edge from `u` to `v` with key `k` while `for (u, v, k, c) in G.edges(data='color', default='red', keys=True):` iterates through all the edges yielding the color attribute with default `'red'` if no color attribute exists.

Edges are returned as tuples with optional data and keys in the order (node, neighbor, key, data). If `keys=True` is not provided, the tuples will just be (node, neighbor, data), but multiple tuples with the same node and neighbor will be generated when multiple edges between two nodes exist.

Parameters

nbunch
: [single node, container, or all nodes (default= all nodes)] The view will only report edges from 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 (u, v, k, d) tuples when data is also requested (the default) and (u, v, k) tuples when data is not requested.

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

edges
[OutMultiEdgeView] A view of edge attributes, usually it iterates over (u, v) (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:

in_edges*, *out_edges

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.MultiDiGraph()
>>> nx.add_path(G, [0, 1, 2])
>>> key = G.add_edge(2, 3, weight=5)
>>> key2 = G.add_edge(1, 2) # second edge between these nodes
>>> [e for e in G.edges()]
[(0, 1), (1, 2), (1, 2), (2, 3)]
>>> list(G.edges(data=True))  # default data is {} (empty dict)
[(0, 1, {}), (1, 2, {}), (1, 2, {}), (2, 3, {'weight': 5})]
>>> list(G.edges(data="weight", default=1))
[(0, 1, 1), (1, 2, 1), (1, 2, 1), (2, 3, 5)]
>>> list(G.edges(keys=True))  # default keys are integers
[(0, 1, 0), (1, 2, 0), (1, 2, 1), (2, 3, 0)]
>>> list(G.edges(data=True, keys=True))
[(0, 1, 0, {}), (1, 2, 0, {}), (1, 2, 1, {}), (2, 3, 0, {'weight': 5})]
>>> list(G.edges(data="weight", default=1, keys=True))
[(0, 1, 0, 1), (1, 2, 0, 1), (1, 2, 1, 1), (2, 3, 0, 5)]
>>> list(G.edges([0, 2]))
[(0, 1), (2, 3)]
>>> list(G.edges(0))
[(0, 1)]
>>> list(G.edges(1))
[(1, 2), (1, 2)]
```

MultiDiGraph.out_edges

property MultiDiGraph.**out_edges**

An OutMultiEdgeView of the Graph as G.edges or G.edges().

edges(self, nbunch=None, data=False, keys=False, default=None)

The OutMultiEdgeView provides set-like operations on the edge-tuples as well as edge attribute lookup. When called, it also provides an EdgeDataView object which allows control of access to edge attributes (but does not provide set-like operations). Hence, `G.edges[u, v, k]['color']` provides the value of the color attribute

for the edge from `u` to `v` with key `k` while `for (u, v, k, c) in G.edges(data='color', default='red', keys=True):` iterates through all the edges yielding the color attribute with default `'red'` if no color attribute exists.

Edges are returned as tuples with optional data and keys in the order (node, neighbor, key, data). If `keys=True` is not provided, the tuples will just be (node, neighbor, data), but multiple tuples with the same node and neighbor will be generated when multiple edges between two nodes exist.

Parameters

nbunch
[single node, container, or all nodes (default= all nodes)] The view will only report edges from 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 (u, v, k, d) tuples when data is also requested (the default) and (u, v, k) tuples when data is not requested.

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

edges
[OutMultiEdgeView] A view of edge attributes, usually it iterates over (u, v) (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:

in_edges, *out_edges*

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.MultiDiGraph()
>>> nx.add_path(G, [0, 1, 2])
>>> key = G.add_edge(2, 3, weight=5)
>>> key2 = G.add_edge(1, 2) # second edge between these nodes
>>> [e for e in G.edges()]
[(0, 1), (1, 2), (1, 2), (2, 3)]
>>> list(G.edges(data=True))  # default data is {} (empty dict)
[(0, 1, {}), (1, 2, {}), (1, 2, {}), (2, 3, {'weight': 5})]
>>> list(G.edges(data="weight", default=1))
[(0, 1, 1), (1, 2, 1), (1, 2, 1), (2, 3, 5)]
>>> list(G.edges(keys=True))  # default keys are integers
[(0, 1, 0), (1, 2, 0), (1, 2, 1), (2, 3, 0)]
>>> list(G.edges(data=True, keys=True))
[(0, 1, 0, {}), (1, 2, 0, {}), (1, 2, 1, {}), (2, 3, 0, {'weight': 5})]
```

(continues on next page)

(continued from previous page)

```
>>> list(G.edges(data="weight", default=1, keys=True))
[(0, 1, 0, 1), (1, 2, 0, 1), (1, 2, 1, 1), (2, 3, 0, 5)]
>>> list(G.edges([0, 2]))
[(0, 1), (2, 3)]
>>> list(G.edges(0))
[(0, 1)]
>>> list(G.edges(1))
[(1, 2), (1, 2)]
```

MultiDiGraph.in_edges

property MultiDiGraph.**in_edges**

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

MultiDiGraph.has_edge

MultiDiGraph.**has_edge**(*u*, *v*, *key=None*)

Returns True if the graph has an edge between nodes u and v.

This is the same as `v in G[u] or key in G[u][v]` without KeyError exceptions.

Parameters

u, v
[nodes] Nodes can be, for example, strings or numbers.

key
[hashable identifier, optional (default=None)] If specified return True only if the edge with key is found.

Returns

edge_ind
[bool] True if edge is in the graph, False otherwise.

Examples

Can be called either using two nodes u, v, an edge tuple (u, v), or an edge tuple (u, v, key).

```
>>> G = nx.MultiGraph()  # or MultiDiGraph
>>> nx.add_path(G, [0, 1, 2, 3])
>>> G.has_edge(0, 1)  # using two nodes
True
>>> e = (0, 1)
>>> G.has_edge(*e)  #  e is a 2-tuple (u, v)
True
>>> G.add_edge(0, 1, key="a")
'a'
>>> G.has_edge(0, 1, key="a")  # specify key
True
>>> G.has_edge(1, 0, key="a")  # edges aren't directed
True
>>> e = (0, 1, "a")
>>> G.has_edge(*e)  # e is a 3-tuple (u, v, 'a')
True
```

The following syntax are equivalent:

```
>>> G.has_edge(0, 1)
True
>>> 1 in G[0]  # though this gives :exc:`KeyError` if 0 not in G
True
>>> 0 in G[1]  # other order; also gives :exc:`KeyError` if 0 not in G
True
```

MultiDiGraph.get_edge_data

MultiDiGraph.**get_edge_data**(*u*, *v*, *key=None*, *default=None*)

Returns the attribute dictionary associated with edge (u, v, key).

If a key is not provided, returns a dictionary mapping edge keys to attribute dictionaries for each edge between u and v.

This is identical to `G[u][v][key]` except the default is returned instead of an exception is the edge doesn't exist.

Parameters

u, v
[nodes]

default
[any Python object (default=None)] Value to return if the specific edge (u, v, key) is not found, OR if there are no edges between u and v and no key is specified.

key
[hashable identifier, optional (default=None)] Return data only for the edge with specified key, as an attribute dictionary (rather than a dictionary mapping keys to attribute dictionaries).

Returns

edge_dict
[dictionary] The edge attribute dictionary, OR a dictionary mapping edge keys to attribute dictionaries for each of those edges if no specific key is provided (even if there's only one edge between u and v).

Examples

```
>>> G = nx.MultiGraph()  # or MultiDiGraph
>>> key = G.add_edge(0, 1, key="a", weight=7)
>>> G[0][1]["a"]  # key='a'
{'weight': 7}
>>> G.edges[0, 1, "a"]  # key='a'
{'weight': 7}
```

Warning: we protect the graph data structure by making `G.edges` and `G[1][2]` read-only dict-like structures. However, you can assign values to attributes in e.g. `G.edges[1, 2, 'a']` or `G[1][2]['a']` using an additional bracket as shown next. You need to specify all edge info to assign to the edge data associated with an edge.

```
>>> G[0][1]["a"]["weight"] = 10
>>> G.edges[0, 1, "a"]["weight"] = 10
>>> G[0][1]["a"]["weight"]
10
>>> G.edges[1, 0, "a"]["weight"]
10
```

```
>>> G = nx.MultiGraph()  # or MultiDiGraph
>>> nx.add_path(G, [0, 1, 2, 3])
>>> G.edges[0, 1, 0]["weight"] = 5
>>> G.get_edge_data(0, 1)
{0: {'weight': 5}}
>>> e = (0, 1)
```

(continues on next page)

(continued from previous page)

```
>>> G.get_edge_data(*e)  # tuple form
{0: {'weight': 5}}
>>> G.get_edge_data(3, 0)  # edge not in graph, returns None
>>> G.get_edge_data(3, 0, default=0)  # edge not in graph, return default
0
>>> G.get_edge_data(1, 0, 0)  # specific key gives back
{'weight': 5}
```

MultiDiGraph.neighbors

MultiDiGraph.**neighbors**(*n*)

Returns an iterator over successor nodes of n.

A successor of n is a node m such that there exists a directed edge from n to m.

Parameters

n
[node] A node in the graph

Raises

NetworkXError
If n is not in the graph.

See also:

predecessors

Notes

neighbors() and successors() are the same.

MultiDiGraph.adj

property MultiDiGraph.**adj**

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-dict. So `G.adj[3][2][0]['color'] = 'blue'` sets the color of the edge `(3, 2, 0)` 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.

MultiDiGraph.__getitem__

MultiDiGraph.**__getitem__**(*n*)

Returns a dict of neighbors of node n. Use: 'G[n]'.

Parameters

n
[node] A node in the graph.

Returns

adj_dict
[dictionary] The adjacency dictionary for nodes connected to n.

Notes

G[n] is the same as G.adj[n] and similar to G.neighbors(n) (which is an iterator over G.adj[n])

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G[0]
AtlasView({1: {}})
```

MultiDiGraph.successors

MultiDiGraph.**successors**(*n*)

Returns an iterator over successor nodes of n.

A successor of n is a node m such that there exists a directed edge from n to m.

Parameters

n
[node] A node in the graph

Raises

NetworkXError
If n is not in the graph.

See also:

predecessors

Notes

neighbors() and successors() are the same.

MultiDiGraph.succ

property MultiDiGraph.**succ**

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 edgekey-dict. So `G.adj[3][2][0]['color'] = 'blue'` sets the color of the edge `(3, 2, 0)` 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.succ` is identical to `G.adj`.

MultiDiGraph.predecessors

MultiDiGraph.**predecessors**(*n*)

Returns an iterator over predecessor nodes of n.

A predecessor of n is a node m such that there exists a directed edge from m to n.

Parameters

n
[node] A node in the graph

Raises

NetworkXError
If n is not in the graph.

See also:

successors

MultiDiGraph.adjacency

MultiDiGraph.**adjacency**()

Returns an iterator over (node, adjacency dict) tuples for all nodes.

For directed graphs, only outgoing neighbors/adjacencies are included.

Returns

adj_iter
[iterator] An iterator over (node, adjacency dictionary) for all nodes in the graph.

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> [(n, nbrdict) for n, nbrdict in G.adjacency()]
[(0, {1: {}}), (1, {0: {}, 2: {}}), (2, {1: {}, 3: {}}), (3, {2: {}})]
```

MultiDiGraph.nbunch_iter

MultiDiGraph.**nbunch_iter**(*nbunch=None*)

Returns an iterator over nodes contained in nbunch that are also in the graph.

The nodes in nbunch are checked for membership in the graph and if not are silently ignored.

Parameters

nbunch
[single node, container, or all nodes (default= all nodes)] The view will only report edges incident to these nodes.

Returns

niter
[iterator] An iterator over nodes in nbunch that are also in the graph. If nbunch is None, iterate over all nodes in the graph.

Raises

NetworkXError
If nbunch is not a node or sequence of nodes. If a node in nbunch is not hashable.

See also:

Graph.__iter__

Notes

When nbunch is an iterator, the returned iterator yields values directly from nbunch, becoming exhausted when nbunch is exhausted.

To test whether nbunch is a single node, one can use "if nbunch in self:", even after processing with this routine.

If nbunch is not a node or a (possibly empty) sequence/iterator or None, a *NetworkXError* is raised. Also, if any object in nbunch is not hashable, a *NetworkXError* is raised.

Counting nodes edges and neighbors

MultiDiGraph.order()	Returns the number of nodes in the graph.
MultiDiGraph.number_of_nodes()	Returns the number of nodes in the graph.
MultiDiGraph.__len__()	Returns the number of nodes in the graph.
MultiDiGraph.degree	A DegreeView for the Graph as G.degree or G.degree().
MultiDiGraph.in_degree	A DegreeView for (node, in_degree) or in_degree for single node.
MultiDiGraph.out_degree	Returns an iterator for (node, out-degree) or out-degree for single node.
MultiDiGraph.size([weight])	Returns the number of edges or total of all edge weights.
MultiDiGraph.number_of_edges([u, v])	Returns the number of edges between two nodes.

MultiDiGraph.order

MultiDiGraph.**order**()

Returns the number of nodes in the graph.

Returns

nnodes
[int] The number of nodes in the graph.

See also:

number_of_nodes
identical method

__len__
identical method

Examples

```
>>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.order()
3
```

MultiDiGraph.number_of_nodes

MultiDiGraph.**number_of_nodes**()

Returns the number of nodes in the graph.

Returns

nnodes
[int] The number of nodes in the graph.

See also:

order
identical method

__len__
identical method

Examples

```
>>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.number_of_nodes()
3
```

MultiDiGraph.__len__

MultiDiGraph.**__len__**()

Returns the number of nodes in the graph. Use: 'len(G)'.

Returns

nnodes
[int] The number of nodes in the graph.

See also:

number_of_nodes
identical method

order
identical method

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> len(G)
4
```

MultiDiGraph.degree

property MultiDiGraph.**degree**

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
>>> list(G.degree([0, 1, 2])) # parallel edges are counted
[(0, 2), (1, 3), (2, 2)]
```

MultiDiGraph.in_degree

property MultiDiGraph.**in_degree**

A DegreeView for (node, in_degree) or in_degree for single node.

The node in-degree is the number of edges pointing in 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 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

If a single node is requested
deg
[int] Degree of the node

OR if multiple nodes are requested
nd_iter
[iterator] The iterator returns two-tuples of (node, in-degree).

See also:

degree, *out_degree*

Examples

```
>>> G = nx.MultiDiGraph()
>>> nx.add_path(G, [0, 1, 2, 3])
>>> G.in_degree(0)  # node 0 with degree 0
0
>>> list(G.in_degree([0, 1, 2]))
[(0, 0), (1, 1), (2, 1)]
>>> G.add_edge(0, 1) # parallel edge
1
>>> list(G.in_degree([0, 1, 2])) # parallel edges counted
[(0, 0), (1, 2), (2, 1)]
```

MultiDiGraph.out_degree

property MultiDiGraph.**out_degree**

Returns an iterator for (node, out-degree) or out-degree for single node.

out_degree(self, nbunch=None, weight=None)

The node out-degree is the number of edges pointing out of the node. This function returns the out-degree for a single node or an iterator for a bunch of nodes or if nothing is passed as argument.

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 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.

Returns

If a single node is requested
deg
[int] Degree of the node

OR if multiple nodes are requested
nd_iter
[iterator] The iterator returns two-tuples of (node, out-degree).

See also:

degree, *in_degree*

Examples

```
>>> G = nx.MultiDiGraph()
>>> nx.add_path(G, [0, 1, 2, 3])
>>> G.out_degree(0)  # node 0 with degree 1
1
>>> list(G.out_degree([0, 1, 2]))
[(0, 1), (1, 1), (2, 1)]
>>> G.add_edge(0, 1) # parallel edge
1
>>> list(G.out_degree([0, 1, 2])) # counts parallel edges
[(0, 2), (1, 1), (2, 1)]
```

MultiDiGraph.size

MultiDiGraph.**size**(*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
```

MultiDiGraph.number_of_edges

MultiDiGraph.**number_of_edges**(*u=None*, *v=None*)

Returns the number of edges between two nodes.

Parameters

u, v
[nodes, optional (Gefault=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 multigraphs, this method counts the total number of edges in the graph:

```
>>> G = nx.MultiGraph()
>>> G.add_edges_from([(0, 1), (0, 1), (1, 2)])
[0, 1, 0]
>>> 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)
2
```

For directed multigraphs, this method can count the total number of directed edges from `u` to `v`:

```
>>> G = nx.MultiDiGraph()
>>> G.add_edges_from([(0, 1), (0, 1), (1, 0)])
[0, 1, 0]
>>> G.number_of_edges(0, 1)
2
>>> G.number_of_edges(1, 0)
1
```

Making copies and subgraphs

MultiDiGraph.copy([as_view])	Returns a copy of the graph.
MultiDiGraph.to_undirected([reciprocal, as_view])	Returns an undirected representation of the digraph.
MultiDiGraph.to_directed([as_view])	Returns a directed representation of the graph.
MultiDiGraph.subgraph(nodes)	Returns a SubGraph view of the subgraph induced on `nodes`.
MultiDiGraph.edge_subgraph(edges)	Returns the subgraph induced by the specified edges.
MultiDiGraph.reverse([copy])	Returns the reverse of the graph.

MultiDiGraph.copy

MultiDiGraph.**copy**(*as_view=False*)

Returns a copy of the graph.

The copy method by default returns an independent shallow copy of the graph and attributes. That is, if an attribute is a container, that container is shared by the original an the copy. Use Python's `copy.deepcopy` for new containers.

If `as_view` is True then a view is returned instead of a copy.

Parameters

as_view
[bool, optional (default=False)] If True, the returned graph-view provides a read-only view of the original graph without actually copying any data.

Returns

G
[Graph] A copy of the graph.

See also:

to_directed
return a directed copy of the graph.

Notes

All copies reproduce the graph structure, but data attributes may be handled in different ways. There are four types of copies of a graph that people might want.

Deepcopy – A "deepcopy" copies the graph structure as well as all data attributes and any objects they might contain. The entire graph object is new so that changes in the copy do not affect the original object. (see Python's copy.deepcopy)

Data Reference (Shallow) – For a shallow copy the graph structure is copied but the edge, node and graph attribute dicts are references to those in the original graph. This saves time and memory but could cause confusion if you change an attribute in one graph and it changes the attribute in the other. NetworkX does not provide this level of shallow copy.

Independent Shallow – This copy creates new independent attribute dicts and then does a shallow copy of the attributes. That is, any attributes that are containers are shared between the new graph and the original. This is exactly what `dict.copy()` provides. You can obtain this style copy using:

```
>>> G = nx.path_graph(5)
>>> H = G.copy()
>>> H = G.copy(as_view=False)
>>> H = nx.Graph(G)
>>> H = G.__class__(G)
```

Fresh Data – For fresh data, the graph structure is copied while new empty data attribute dicts are created. The resulting graph is independent of the original and it has no edge, node or graph attributes. Fresh copies are not enabled. Instead use:

```
>>> H = G.__class__()
>>> H.add_nodes_from(G)
>>> H.add_edges_from(G.edges)
```

View – Inspired by dict-views, graph-views act like read-only versions of the original graph, providing a copy of the original structure without requiring any memory for copying the information.

See the Python copy module for more information on shallow and deep copies, https://docs.python.org/3/library/copy.html.

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> H = G.copy()
```

MultiDiGraph.to_undirected

`MultiDiGraph.`**`to_undirected`**(*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
: [MultiGraph] 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:

MultiGraph, ***copy***, ***add_edge***, ***add_edges_from***

Notes

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 D=MultiDiGraph(G) 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 MultiDiGraph to use dict-like objects in the data structure, those changes do not transfer to the MultiGraph 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)]
```

MultiDiGraph.to_directed

`MultiDiGraph.`**`to_directed`**(*as_view=False*)

Returns a directed representation of the graph.

Returns

G

[MultiDiGraph] A directed graph with the same name, same nodes, and with each edge (u, v, k, data) replaced by two directed edges (u, v, k, data) and (v, u, k, data).

Notes

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 D=MultiDiGraph(G) 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 MultiGraph to use dict-like objects in the data structure, those changes do not transfer to the MultiDiGraph created by this method.

Examples

```
>>> G = nx.MultiGraph()
>>> G.add_edge(0, 1)
0
>>> G.add_edge(0, 1)
1
>>> H = G.to_directed()
>>> list(H.edges)
[(0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1)]
```

If already directed, return a (deep) copy

```
>>> G = nx.MultiDiGraph()
>>> G.add_edge(0, 1)
0
>>> H = G.to_directed()
>>> list(H.edges)
[(0, 1, 0)]
```

MultiDiGraph.subgraph

MultiDiGraph.**subgraph**(*nodes*)

Returns a SubGraph view of the subgraph induced on *nodes*.

The induced subgraph of the graph contains the nodes in *nodes* and the edges between those nodes.

Parameters

nodes
[list, iterable] A container of nodes which will be iterated through once.

Returns

G
[SubGraph View] A subgraph view of the graph. The graph structure cannot be changed but node/edge attributes can and are shared with the original graph.

Notes

The graph, edge and node attributes are shared with the original graph. Changes to the graph structure is ruled out by the view, but changes to attributes are reflected in the original graph.

To create a subgraph with its own copy of the edge/node attributes use: G.subgraph(nodes).copy()

For an inplace reduction of a graph to a subgraph you can remove nodes: G.remove_nodes_from([n for n in G if n not in set(nodes)])

Subgraph views are sometimes NOT what you want. In most cases where you want to do more than simply look at the induced edges, it makes more sense to just create the subgraph as its own graph with code like:

```
# Create a subgraph SG based on a (possibly multigraph) G
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)
```

(continues on next page)

(continued from previous page)

```
        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)]
```

MultiDiGraph.edge_subgraph

MultiDiGraph.**edge_subgraph**(*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)]
```

MultiDiGraph.reverse

MultiDiGraph.**reverse**(*copy=True*)

Returns the reverse of the graph.

The reverse is a graph with the same nodes and edges but with the directions of the edges reversed.

Parameters

copy
[bool optional (default=True)] If True, return a new DiGraph holding the reversed edges. If False, the reverse graph is created using a view of the original graph.

Note: NetworkX uses `dicts` to store the nodes and neighbors in a graph. So the reporting of nodes and edges for the base graph classes may not necessarily be consistent across versions and platforms; however, the reporting for CPython is consistent across platforms and versions after 3.6.

2.3 Graph Views

View of Graphs as SubGraph, Reverse, Directed, Undirected.

In some algorithms it is convenient to temporarily morph a graph to exclude some nodes or edges. It should be better to do that via a view than to remove and then re-add. In other algorithms it is convenient to temporarily morph a graph to reverse directed edges, or treat a directed graph as undirected, etc. This module provides those graph views.

The resulting views are essentially read-only graphs that report data from the orignal graph object. We provide an attribute G._graph which points to the underlying graph object.

Note: Since graphviews look like graphs, one can end up with view-of-view-of-view chains. Be careful with chains because they become very slow with about 15 nested views. For the common simple case of node induced subgraphs created from the graph class, we short-cut the chain by returning a subgraph of the original graph directly rather than a subgraph of a subgraph. We are careful not to disrupt any edge filter in the middle subgraph. In general, determining how to short-cut the chain is tricky and much harder with restricted_views than with induced subgraphs. Often it is easiest to use .copy() to avoid chains.

generic_graph_view(G[, create_using])	
subgraph_view(G[, filter_node, filter_edge])	View of `G` applying a filter on nodes and edges.
reverse_view(G)	View of `G` with edge directions reversed

2.3.1 generic_graph_view

generic_graph_view(*G*, *create_using=None*)

2.3.2 subgraph_view

subgraph_view(*G*, *filter_node=<function no_filter>*, *filter_edge=<function no_filter>*)

View of `G` applying a filter on nodes and edges.

subgraph_view provides a read-only view of the input graph that excludes nodes and edges based on the outcome of two filter functions `filter_node` and `filter_edge`.

The `filter_node` function takes one argument — the node — and returns `True` if the node should be included in the subgraph, and `False` if it should not be included.

The `filter_edge` function takes two (or three arguments if `G` is a multi-graph) — the nodes describing an edge, plus the edge-key if parallel edges are possible — and returns `True` if the edge should be included in the subgraph, and `False` if it should not be included.

Both node and edge filter functions are called on graph elements as they are queried, meaning there is no up-front cost to creating the view.

Parameters

G
[networkx.Graph] A directed/undirected graph/multigraph

filter_node
[callable, optional] A function taking a node as input, which returns `True` if the node should appear in the view.

filter_edge
[callable, optional] A function taking as input the two nodes describing an edge (plus the edge-key if `G` is a multi-graph), which returns `True` if the edge should appear in the view.

Returns

graph
[networkx.Graph] A read-only graph view of the input graph.

Examples

```
>>> G = nx.path_graph(6)
```

Filter functions operate on the node, and return `True` if the node should appear in the view:

```
>>> def filter_node(n1):
...     return n1 != 5
...
>>> view = nx.subgraph_view(G, filter_node=filter_node)
>>> view.nodes()
NodeView((0, 1, 2, 3, 4))
```

We can use a closure pattern to filter graph elements based on additional data — for example, filtering on edge data attached to the graph:

```
>>> G[3][4]["cross_me"] = False
>>> def filter_edge(n1, n2):
...     return G[n1][n2].get("cross_me", True)
...
>>> view = nx.subgraph_view(G, filter_edge=filter_edge)
>>> view.edges()
EdgeView([(0, 1), (1, 2), (2, 3), (4, 5)])
```

```
>>> view = nx.subgraph_view(G, filter_node=filter_node, filter_edge=filter_edge,)
>>> view.nodes()
NodeView((0, 1, 2, 3, 4))
>>> view.edges()
EdgeView([(0, 1), (1, 2), (2, 3)])
```

2.3.3 reverse_view

reverse_view(*G*)

View of G with edge directions reversed

reverse_view returns a read-only view of the input graph where edge directions are reversed.

Identical to digraph.reverse(copy=False)

Parameters

G
[networkx.DiGraph]

Returns

graph
[networkx.DiGraph]

Examples

```
>>> G = nx.DiGraph()
>>> G.add_edge(1, 2)
>>> G.add_edge(2, 3)
>>> G.edges()
OutEdgeView([(1, 2), (2, 3)])
```

```
>>> view = nx.reverse_view(G)
>>> view.edges()
OutEdgeView([(2, 1), (3, 2)])
```

2.4 Core Views

Views of core data structures such as nested Mappings (e.g. dict-of-dicts). These Views often restrict element access, with either the entire view or layers of nested mappings being read-only.

AtlasView(d)	An AtlasView is a Read-only Mapping of Mappings.
AdjacencyView(d)	An AdjacencyView is a Read-only Map of Maps of Maps.
MultiAdjacencyView(d)	An MultiAdjacencyView is a Read-only Map of Maps of Maps of Maps.
UnionAtlas(succ, pred)	A read-only union of two atlases (dict-of-dict).
UnionAdjacency(succ, pred)	A read-only union of dict Adjacencies as a Map of Maps of Maps.
UnionMultiInner(succ, pred)	A read-only union of two inner dicts of MultiAdjacencies.
UnionMultiAdjacency(succ, pred)	A read-only union of two dict MultiAdjacencies.
FilterAtlas(d, NODE_OK)	
FilterAdjacency(d, NODE_OK, EDGE_OK)	
FilterMultiInner(d, NODE_OK, EDGE_OK)	
FilterMultiAdjacency(d, NODE_OK, EDGE_OK)	

2.4.1 networkx.classes.coreviews.AtlasView

class AtlasView(*d*)

An AtlasView is a Read-only Mapping of Mappings.

It is a View into a dict-of-dict data structure. The inner level of dict is read-write. But the outer level is read-only.

See also:

AdjacencyView
View into dict-of-dict-of-dict

MultiAdjacencyView
View into dict-of-dict-of-dict-of-dict

__init__(*d*)

Methods

copy()
get(k[,d])
items()
keys()
values()

AtlasView.copy

AtlasView.**copy**()

AtlasView.get

AtlasView.**get**(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

AtlasView.items

AtlasView.**items**() → a set-like object providing a view on D's items

AtlasView.keys

AtlasView.**keys**() → a set-like object providing a view on D's keys

AtlasView.values

AtlasView.**values**() → an object providing a view on D's values

2.4.2 networkx.classes.coreviews.AdjacencyView

class AdjacencyView(*d*)

An AdjacencyView is a Read-only Map of Maps of Maps.

It is a View into a dict-of-dict-of-dict data structure. The inner level of dict is read-write. But the outer levels are read-only.

See also:

AtlasView
View into dict-of-dict

MultiAdjacencyView
View into dict-of-dict-of-dict-of-dict

__init__(*d*)

Methods

copy()
get(k[,d])
items()
keys()
values()

AdjacencyView.copy

AdjacencyView.**copy**()

AdjacencyView.get

AdjacencyView.**get**(k[, d]) → D[k] if k in D, else d. d defaults to None.

AdjacencyView.items

AdjacencyView.**items**() → a set-like object providing a view on D's items

AdjacencyView.keys

AdjacencyView.**keys**() → a set-like object providing a view on D's keys

AdjacencyView.values

AdjacencyView.**values**() → an object providing a view on D's values

2.4.3 networkx.classes.coreviews.MultiAdjacencyView

class MultiAdjacencyView(d)

An MultiAdjacencyView is a Read-only Map of Maps of Maps of Maps.

It is a View into a dict-of-dict-of-dict-of-dict data structure. The inner level of dict is read-write. But the outer levels are read-only.

See also:

AtlasView
View into dict-of-dict

AdjacencyView
View into dict-of-dict-of-dict

__init__(*d*)

Methods

copy()
get(k[,d])
items()
keys()
values()

MultiAdjacencyView.copy

MultiAdjacencyView.**copy**()

MultiAdjacencyView.get

MultiAdjacencyView.**get**(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

MultiAdjacencyView.items

MultiAdjacencyView.**items**() → a set-like object providing a view on D's items

MultiAdjacencyView.keys

MultiAdjacencyView.**keys**() → a set-like object providing a view on D's keys

MultiAdjacencyView.values

MultiAdjacencyView.**values**() → an object providing a view on D's values

2.4.4 networkx.classes.coreviews.UnionAtlas

class UnionAtlas(*succ*, *pred*)

A read-only union of two atlases (dict-of-dict).

The two dict-of-dicts represent the inner dict of an Adjacency: `G.succ[node]` and `G.pred[node]`. The inner level of dict of both hold attribute key:value pairs and is read-write. But the outer level is read-only.

See also:

UnionAdjacency
View into dict-of-dict-of-dict

UnionMultiAdjacency
View into dict-of-dict-of-dict-of-dict

__init__(*succ*, *pred*)

Methods

copy()
get(k[,d])
items()
keys()
values()

UnionAtlas.copy

UnionAtlas.**copy**()

UnionAtlas.get

UnionAtlas.**get**(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

UnionAtlas.items

UnionAtlas.**items**() → a set-like object providing a view on D's items

UnionAtlas.keys

UnionAtlas.**keys**() → a set-like object providing a view on D's keys

UnionAtlas.values

UnionAtlas.**values**() → an object providing a view on D's values

2.4.5 networkx.classes.coreviews.UnionAdjacency

class UnionAdjacency(*succ*, *pred*)

A read-only union of dict Adjacencies as a Map of Maps of Maps.

The two input dict-of-dict-of-dicts represent the union of `G.succ` and `G.pred`. Return values are UnionAtlas The inner level of dict is read-write. But the middle and outer levels are read-only.

succ : a dict-of-dict-of-dict {node: nbrdict} pred : a dict-of-dict-of-dict {node: nbrdict} The keys for the two dicts should be the same

See also:

UnionAtlas
View into dict-of-dict

UnionMultiAdjacency
View into dict-of-dict-of-dict-of-dict

__init__(*succ*, *pred*)

Methods

copy()
get(k[,d])
items()
keys()
values()

UnionAdjacency.copy

UnionAdjacency.**copy**()

UnionAdjacency.get

UnionAdjacency.**get**(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

UnionAdjacency.items

`UnionAdjacency.`**`items`**`()` → a set-like object providing a view on D's items

UnionAdjacency.keys

`UnionAdjacency.`**`keys`**`()` → a set-like object providing a view on D's keys

UnionAdjacency.values

`UnionAdjacency.`**`values`**`()` → an object providing a view on D's values

2.4.6 networkx.classes.coreviews.UnionMultiInner

class UnionMultiInner(*succ*, *pred*)

A read-only union of two inner dicts of MultiAdjacencies.

The two input dict-of-dict-of-dicts represent the union of `G.succ[node]` and `G.pred[node]` for MultiDiGraphs. Return values are UnionAtlas. The inner level of dict is read-write. But the outer levels are read-only.

See also:

UnionAtlas
View into dict-of-dict

UnionAdjacency
View into dict-of-dict-of-dict

UnionMultiAdjacency
View into dict-of-dict-of-dict-of-dict

__init__(*succ*, *pred*)

Methods

copy()
get(k[,d])
items()
keys()
values()

UnionMultiInner.copy

UnionMultiInner.**copy**()

UnionMultiInner.get

UnionMultiInner.**get**(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

UnionMultiInner.items

UnionMultiInner.**items**() → a set-like object providing a view on D's items

UnionMultiInner.keys

UnionMultiInner.**keys**() → a set-like object providing a view on D's keys

UnionMultiInner.values

UnionMultiInner.**values**() → an object providing a view on D's values

2.4.7 networkx.classes.coreviews.UnionMultiAdjacency

class UnionMultiAdjacency(*succ*, *pred*)

A read-only union of two dict MultiAdjacencies.

The two input dict-of-dict-of-dict-of-dicts represent the union of `G.succ` and `G.pred` for MultiDiGraphs. Return values are UnionAdjacency. The inner level of dict is read-write. But the outer levels are read-only.

See also:

UnionAtlas
View into dict-of-dict

UnionMultiInner
View into dict-of-dict-of-dict

__init__(*succ*, *pred*)

Methods

copy()
get(k[,d])
items()
keys()
values()

UnionMultiAdjacency.copy

UnionMultiAdjacency.**copy**()

UnionMultiAdjacency.get

UnionMultiAdjacency.**get**(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

UnionMultiAdjacency.items

UnionMultiAdjacency.**items**() → a set-like object providing a view on D's items

UnionMultiAdjacency.keys

UnionMultiAdjacency.**keys**() → a set-like object providing a view on D's keys

UnionMultiAdjacency.values

UnionMultiAdjacency.**values**() → an object providing a view on D's values

2.4.8 networkx.classes.coreviews.FilterAtlas

class FilterAtlas(*d*, *NODE_OK*)

__init__(*d*, *NODE_OK*)

Methods

get(k[,d])
items()
keys()
values()

FilterAtlas.get

FilterAtlas.**get**(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

FilterAtlas.items

FilterAtlas.**items**() → a set-like object providing a view on D's items

FilterAtlas.keys

FilterAtlas.**keys**() → a set-like object providing a view on D's keys

FilterAtlas.values

FilterAtlas.**values**() → an object providing a view on D's values

2.4.9 networkx.classes.coreviews.FilterAdjacency

class FilterAdjacency(*d*, *NODE_OK*, *EDGE_OK*)

__init__(*d*, *NODE_OK*, *EDGE_OK*)

Methods

get(k[,d])
items()
keys()
values()

FilterAdjacency.get

`FilterAdjacency.`**`get`**`(`*k*[, *d*]`)` → D[k] if k in D, else d. d defaults to None.

FilterAdjacency.items

`FilterAdjacency.`**`items`**`()` → a set-like object providing a view on D's items

FilterAdjacency.keys

`FilterAdjacency.`**`keys`**`()` → a set-like object providing a view on D's keys

FilterAdjacency.values

`FilterAdjacency.`**`values`**`()` → an object providing a view on D's values

2.4.10 networkx.classes.coreviews.FilterMultiInner

`class FilterMultiInner``(`*d*, *NODE_OK*, *EDGE_OK*`)`

`__init__``(`*d*, *NODE_OK*, *EDGE_OK*`)`

Methods

get(k[,d])
items()
keys()
values()

FilterMultiInner.get

`FilterMultiInner.`**`get`**`(`*k*[, *d*]`)` → D[k] if k in D, else d. d defaults to None.

FilterMultiInner.items

`FilterMultiInner.`**`items`**`()` → a set-like object providing a view on D's items

FilterMultiInner.keys

`FilterMultiInner.`**`keys`**`()` → a set-like object providing a view on D's keys

FilterMultiInner.values

`FilterMultiInner.`**`values`**`()` → an object providing a view on D's values

2.4.11 networkx.classes.coreviews.FilterMultiAdjacency

class FilterMultiAdjacency(*d*, *NODE_OK*, *EDGE_OK*)

__init__(*d*, *NODE_OK*, *EDGE_OK*)

Methods

get(k[,d])
items()
keys()
values()

FilterMultiAdjacency.get

`FilterMultiAdjacency.`**`get`**(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

FilterMultiAdjacency.items

`FilterMultiAdjacency.`**`items`**`()` → a set-like object providing a view on D's items

FilterMultiAdjacency.keys

FilterMultiAdjacency.**keys**() → a set-like object providing a view on D's keys

FilterMultiAdjacency.values

FilterMultiAdjacency.**values**() → an object providing a view on D's values

2.5 Filters

Note: Filters can be used with views to restrict the view (or expand it). They can filter nodes or filter edges. These examples are intended to help you build new ones. They may instead contain all the filters you ever need.

Filter factories to hide or show sets of nodes and edges.

These filters return the function used when creating SubGraph.

no_filter(*items)

hide_nodes(nodes)

hide_edges(edges)

hide_diedges(edges)

hide_multidiedges(edges)

hide_multiedges(edges)

show_nodes(nodes)

show_edges(edges)

show_diedges(edges)

show_multidiedges(edges)

show_multiedges(edges)

2.5.1 no_filter

no_filter(**items*)

2.5.2 hide_nodes

hide_nodes(*nodes*)

2.5.3 hide_edges

hide_edges(*edges*)

2.5.4 hide_diedges

hide_diedges(*edges*)

2.5.5 hide_multidiedges

hide_multidiedges(*edges*)

2.5.6 hide_multiedges

hide_multiedges(*edges*)

2.5.7 networkx.classes.filters.show_nodes

class show_nodes(*nodes*)

__init__(*nodes*)

Methods

2.5.8 show_edges

show_edges(*edges*)

2.5.9 show_diedges

show_diedges(*edges*)

2.5.10 show_multidiedges

show_multidiedges(*edges*)

2.5.11 show_multiedges

show_multiedges(*edges*)

2.6 Backends

Note: This is an experimental feature to dispatch your computations to an alternate backend like GraphBLAS, instead of using pure Python dictionaries for computation. Things will change and break in the future!

Code to support various backends in a plugin dispatch architecture.

2.6.1 Create a Dispatcher

To be a valid plugin, a package must register an entry_point of `networkx.plugins` with a key pointing to the handler.

For example:

` entry_points={'networkx.plugins': 'sparse = networkx_plugin_sparse'} `

The plugin must create a Graph-like object which contains an attribute `__networkx_plugin__` with a value of the entry point name.

Continuing the example above:

``` class WrappedSparse:

    __networkx_plugin__ = "sparse" …

```

When a dispatchable NetworkX algorithm encounters a Graph-like object with a `__networkx_plugin__` attribute, it will look for the associated dispatch object in the entry_points, load it, and dispatch the work to it.

2.6.2 Testing

To assist in validating the backend algorithm implementations, if an environment variable `NETWORKX_GRAPH_CONVERT` is set to a registered plugin keys, the dispatch machinery will automatically convert regular networkx Graphs and DiGraphs to the backend equivalent by calling `<backend dispatcher>.convert_from_nx(G, weight=weight, name=name)`.

The converted object is then passed to the backend implementation of the algorithm. The result is then passed to `<backend dispatcher>.convert_to_nx(result, name=name)` to convert back to a form expected by the NetworkX tests.

By defining `convert_from_nx` and `convert_to_nx` methods and setting the environment variable, NetworkX will automatically route tests on dispatchable algorithms to the backend, allowing the full networkx test suite to be run against the backend implementation.

Example pytest invocation: NETWORKX_GRAPH_CONVERT=sparse pytest –pyargs networkx

Dispatchable algorithms which are not implemented by the backend will cause a `pytest.xfail()`, giving some indication that not all tests are working, while avoiding causing an explicit failure.

A special `on_start_tests(items)` function may be defined by the backend. It will be called with the list of NetworkX tests discovered. Each item is a pytest.Node object. If the backend does not support the test, that test can be marked as xfail.

_dispatch([func, name])	Dispatches to a backend algorithm when the first argument is a backend graph-like object.

2.6.3 _dispatch

_dispatch(*func=None*, *, *name=None*)

Dispatches to a backend algorithm when the first argument is a backend graph-like object.

CHAPTER

THREE

ALGORITHMS

3.1 Approximations and Heuristics

Approximations of graph properties and Heuristic methods for optimization.

Warning: These functions are not imported in the top-level of `networkx`

These functions can be accessed using `networkx.approximation.function_name`

They can be imported using `from networkx.algorithms import approximation` or `from networkx.algorithms.approximation import function_name`

3.1.1 Connectivity

Fast approximation for node connectivity

all_pairs_node_connectivity(G[, nbunch, cutoff])	Compute node connectivity between all pairs of nodes.
local_node_connectivity(G, source, target[, ...])	Compute node connectivity between source and target.
node_connectivity(G[, s, t])	Returns an approximation for node connectivity for a graph or digraph G.

all_pairs_node_connectivity

all_pairs_node_connectivity(*G*, *nbunch=None*, *cutoff=None*)

Compute node connectivity between all pairs of nodes.

Pairwise or local node connectivity between two distinct and nonadjacent nodes is the minimum number of nodes that must be removed (minimum separating cutset) to disconnect them. By Menger's theorem, this is equal to the number of node independent paths (paths that share no nodes other than source and target). Which is what we compute in this function.

This algorithm is a fast approximation that gives an strict lower bound on the actual number of node independent paths between two nodes [1]. It works for both directed and undirected graphs.

Parameters

G

[NetworkX graph]

nbunch: container
Container of nodes. If provided node connectivity will be computed only over pairs of nodes in nbunch.

cutoff
[integer] Maximum node connectivity to consider. If None, the minimum degree of source or target is used as a cutoff in each pair of nodes. Default value None.

Returns

K
[dictionary] Dictionary, keyed by source and target, of pairwise node connectivity

See also:

local_node_connectivity
node_connectivity

References

[1]

Examples

A 3 node cycle with one extra node attached has connectivity 2 between all nodes in the cycle and connectivity 1 between the extra node and the rest:

```
>>> G = nx.cycle_graph(3)
>>> G.add_edge(2, 3)
>>> import pprint  # for nice dictionary formatting
>>> pprint.pprint(nx.all_pairs_node_connectivity(G))
{0: {1: 2, 2: 2, 3: 1},
 1: {0: 2, 2: 2, 3: 1},
 2: {0: 2, 1: 2, 3: 1},
 3: {0: 1, 1: 1, 2: 1}}
```

local_node_connectivity

local_node_connectivity(*G*, *source*, *target*, *cutoff=None*)

Compute node connectivity between source and target.

Pairwise or local node connectivity between two distinct and nonadjacent nodes is the minimum number of nodes that must be removed (minimum separating cutset) to disconnect them. By Menger's theorem, this is equal to the number of node independent paths (paths that share no nodes other than source and target). Which is what we compute in this function.

This algorithm is a fast approximation that gives an strict lower bound on the actual number of node independent paths between two nodes [1]. It works for both directed and undirected graphs.

Parameters

G
[NetworkX graph]

source
[node] Starting node for node connectivity

target
[node] Ending node for node connectivity

cutoff
[integer] Maximum node connectivity to consider. If None, the minimum degree of source or target is used as a cutoff. Default value None.

Returns

k: integer
pairwise node connectivity

See also:

all_pairs_node_connectivity
node_connectivity

Notes

This algorithm [1] finds node independents paths between two nodes by computing their shortest path using BFS, marking the nodes of the path found as 'used' and then searching other shortest paths excluding the nodes marked as used until no more paths exist. It is not exact because a shortest path could use nodes that, if the path were longer, may belong to two different node independent paths. Thus it only guarantees an strict lower bound on node connectivity.

Note that the authors propose a further refinement, losing accuracy and gaining speed, which is not implemented yet.

References

[1]

Examples

```
>>> # Platonic octahedral graph has node connectivity 4
>>> # for each non adjacent node pair
>>> from networkx.algorithms import approximation as approx
>>> G = nx.octahedral_graph()
>>> approx.local_node_connectivity(G, 0, 5)
4
```

node_connectivity

node_connectivity(*G*, *s=None*, *t=None*)

Returns an approximation for node connectivity for a graph or digraph G.

Node connectivity is equal to the minimum number of nodes that must be removed to disconnect G or render it trivial. By Menger's theorem, this is equal to the number of node independent paths (paths that share no nodes other than source and target).

If source and target nodes are provided, this function returns the local node connectivity: the minimum number of nodes that must be removed to break all paths from source to target in G.

This algorithm is based on a fast approximation that gives an strict lower bound on the actual number of node independent paths between two nodes [1]. It works for both directed and undirected graphs.

Parameters

G
[NetworkX graph] Undirected graph

s
[node] Source node. Optional. Default value: None.

t
[node] Target node. Optional. Default value: None.

Returns

K
[integer] Node connectivity of G, or local node connectivity if source and target are provided.

See also:

all_pairs_node_connectivity
local_node_connectivity

Notes

This algorithm [1] finds node independents paths between two nodes by computing their shortest path using BFS, marking the nodes of the path found as 'used' and then searching other shortest paths excluding the nodes marked as used until no more paths exist. It is not exact because a shortest path could use nodes that, if the path were longer, may belong to two different node independent paths. Thus it only guarantees an strict lower bound on node connectivity.

References

[1]

Examples

```
>>> # Platonic octahedral graph is 4-node-connected
>>> from networkx.algorithms import approximation as approx
>>> G = nx.octahedral_graph()
>>> approx.node_connectivity(G)
4
```

3.1.2 K-components

Fast approximation for k-component structure

k_components(G[, min_density])	Returns the approximate k-component structure of a graph G.

k_components

k_components(*G*, *min_density=0.95*)

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.

See also:

k_components

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.

References

[1], [2], [3]

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)
```

3.1.3 Clique

Functions for computing large cliques and maximum independent sets.

maximum_independent_set(G)	Returns an approximate maximum independent set.
max_clique(G)	Find the Maximum Clique
clique_removal(G)	Repeatedly remove cliques from the graph.
large_clique_size(G)	Find the size of a large clique in a graph.

maximum_independent_set

maximum_independent_set(*G*)

Returns an approximate maximum independent set.

Independent set or stable set is a set of vertices in a graph, no two of which are adjacent. That is, it is a set I of vertices such that for every two vertices in I, there is no edge connecting the two. Equivalently, each edge in the graph has at most one endpoint in I. The size of an independent set is the number of vertices it contains [1].

A maximum independent set is a largest independent set for a given graph G and its size is denoted $\alpha(G)$. The problem of finding such a set is called the maximum independent set problem and is an NP-hard optimization problem. As such, it is unlikely that there exists an efficient algorithm for finding a maximum independent set of a graph.

The Independent Set algorithm is based on [2].

Parameters

G
[NetworkX graph] Undirected graph

Returns

iset
[Set] The apx-maximum independent set

Raises

NetworkXNotImplemented
If the graph is directed or is a multigraph.

Notes

Finds the $O(|V|/(log|V|)^2)$ apx of independent set in the worst case.

References

[1], [2]

max_clique

max_clique(G)

Find the Maximum Clique

Finds the $O(|V|/(log|V|)^2)$ apx of maximum clique/independent set in the worst case.

Parameters

G
[NetworkX graph] Undirected graph

Returns

clique
[set] The apx-maximum clique of the graph

Raises

NetworkXNotImplemented
If the graph is directed or is a multigraph.

Notes

A clique in an undirected graph G = (V, E) is a subset of the vertex set `C subseteq V` such that for every two vertices in C there exists an edge connecting the two. This is equivalent to saying that the subgraph induced by C is complete (in some cases, the term clique may also refer to the subgraph).

A maximum clique is a clique of the largest possible size in a given graph. The clique number `omega(G)` of a graph G is the number of vertices in a maximum clique in G. The intersection number of G is the smallest number of cliques that together cover all edges of G.

https://en.wikipedia.org/wiki/Maximum_clique

References

[1]

clique_removal

clique_removal(*G*)

Repeatedly remove cliques from the graph.

Results in a $O(|V|/(\log|V|)^2)$ approximation of maximum clique and independent set. Returns the largest independent set found, along with found maximal cliques.

Parameters

G
[NetworkX graph] Undirected graph

Returns

max_ind_cliques
[(set, list) tuple] 2-tuple of Maximal Independent Set and list of maximal cliques (sets).

Raises

NetworkXNotImplemented
If the graph is directed or is a multigraph.

References

[1]

large_clique_size

large_clique_size(*G*)

Find the size of a large clique in a graph.

A *clique* is a subset of nodes in which each pair of nodes is adjacent. This function is a heuristic for finding the size of a large clique in the graph.

Parameters

G
[NetworkX graph]

Returns

k: integer
The size of a large clique in the graph.

Raises

NetworkXNotImplemented
If the graph is directed or is a multigraph.

See also:

networkx.algorithms.approximation.clique.max_clique()
A function that returns an approximate maximum clique with a guarantee on the approximation ratio.

networkx.algorithms.clique
Functions for finding the exact maximum clique in a graph.

Notes

This implementation is from [1]. Its worst case time complexity is $O(nd^2)$, where n is the number of nodes in the graph and d is the maximum degree.

This function is a heuristic, which means it may work well in practice, but there is no rigorous mathematical guarantee on the ratio between the returned number and the actual largest clique size in the graph.

References

[1]

3.1.4 Clustering

average_clustering(G[, trials, seed])	Estimates the average clustering coefficient of G.

average_clustering

average_clustering(*G*, *trials=1000*, *seed=None*)

Estimates the average clustering coefficient of G.

The local clustering of each node in `G` is the fraction of triangles that actually exist over all possible triangles in its neighborhood. The average clustering coefficient of a graph `G` is the mean of local clusterings.

This function finds an approximate average clustering coefficient for G by repeating `n` times (defined in `trials`) the following experiment: choose a node at random, choose two of its neighbors at random, and check if they are connected. The approximate coefficient is the fraction of triangles found over the number of trials [1].

Parameters

G
[NetworkX graph]

trials
[integer] Number of trials to perform (default 1000).

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

c
[float] Approximated average clustering coefficient.

References

[1]

Examples

```
>>> from networkx.algorithms import approximation
>>> G = nx.erdos_renyi_graph(10, 0.2, seed=10)
>>> approximation.average_clustering(G, trials=1000, seed=10)
0.214
```

3.1.5 Distance Measures

Distance measures approximated metrics.

diameter(G[, seed])	Returns a lower bound on the diameter of the graph G.

diameter

diameter(*G*, *seed=None*)

Returns a lower bound on the diameter of the graph G.

The function computes a lower bound on the diameter (i.e., the maximum eccentricity) of a directed or undirected graph G. The procedure used varies depending on the graph being directed or not.

If G is an `undirected` graph, then the function uses the `2-sweep` algorithm [1]. The main idea is to pick the farthest node from a random node and return its eccentricity.

Otherwise, if G is a `directed` graph, the function uses the `2-dSweep` algorithm [2], The procedure starts by selecting a random source node s from which it performs a forward and a backward BFS. Let a_1 and a_2 be the farthest nodes in the forward and backward cases, respectively. Then, it computes the backward eccentricity of a_1 using a backward BFS and the forward eccentricity of a_2 using a forward BFS. Finally, it returns the best lower bound between the two.

In both cases, the time complexity is linear with respect to the size of G.

Parameters

G
[NetworkX graph]

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

d
[integer] Lower Bound on the Diameter of G

Raises

NetworkXError
If the graph is empty or If the graph is undirected and not connected or If the graph is directed and not strongly connected.

See also:

networkx.algorithms.distance_measures.diameter

References

[1], [2]

3.1.6 Dominating Set

Functions for finding node and edge dominating sets.

A dominating set for an undirected graph *G* with vertex set *V* and edge set *E* is a subset *D* of *V* such that every vertex not in *D* is adjacent to at least one member of *D*. An edge dominating set is a subset *F* of *E* such that every edge not in *F* is incident to an endpoint of at least one edge in *F*.

min_weighted_dominating_set(G[, weight])	Returns a dominating set that approximates the minimum weight node dominating set.
min_edge_dominating_set(G)	Returns minimum cardinality edge dominating set.

min_weighted_dominating_set

min_weighted_dominating_set(*G*, *weight=None*)

Returns a dominating set that approximates the minimum weight node dominating set.

Parameters

G
[NetworkX graph] Undirected graph.

weight
[string] The node attribute storing the weight of an node. If provided, the node attribute with this key must be a number for each node. If not provided, each node is assumed to have weight one.

Returns

min_weight_dominating_set
[set] A set of nodes, the sum of whose weights is no more than `(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.

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]

min_edge_dominating_set

min_edge_dominating_set(G)

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.

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 $O(|E|)$.

3.1.7 Matching

Given a graph G = (V,E), a matching M in G is a set of pairwise non-adjacent edges; that is, no two edges share a common vertex.

Wikipedia: Matching

min_maximal_matching(G)	Returns the minimum maximal matching of G.

min_maximal_matching

min_maximal_matching(G)

Returns the minimum maximal matching of G. That is, out of all maximal matchings of the graph G, the smallest is returned.

Parameters

G
[NetworkX graph] Undirected graph

Returns

min_maximal_matching
[set] Returns a set of edges such that no two edges share a common endpoint and every edge not in the set shares some common endpoint in the set. Cardinality will be 2*OPT in the worst case.

Notes

The algorithm computes an approximate solution fo the minimum maximal cardinality matching problem. The solution is no more than 2 * OPT in size. Runtime is $O(|E|)$.

References

[1]

3.1.8 Ramsey

Ramsey numbers.

ramsey_R2(G)	Compute the largest clique and largest independent set in G.

ramsey_R2

ramsey_R2(*G*)

Compute the largest clique and largest independent set in G.

This can be used to estimate bounds for the 2-color Ramsey number R(2;s,t) for G.

This is a recursive implementation which could run into trouble for large recursions. Note that self-loop edges are ignored.

Parameters

G
[NetworkX graph] Undirected graph

Returns

max_pair
[(set, set) tuple] Maximum clique, Maximum independent set.

Raises

NetworkXNotImplemented
If the graph is directed or is a multigraph.

3.1.9 Steiner Tree

metric_closure(G[, weight])	Return the metric closure of a graph.
steiner_tree(G, terminal_nodes[, weight, method])	Return an approximation to the minimum Steiner tree of a graph.

metric_closure

metric_closure(*G*, *weight='weight'*)

Return the metric closure of a graph.

The metric closure of a graph *G* is the complete graph in which each edge is weighted by the shortest path distance between the nodes in *G* .

Parameters

G
[NetworkX graph]

Returns

NetworkX graph
Metric closure of the graph `G`.

steiner_tree

steiner_tree(*G*, *terminal_nodes*, *weight='weight'*, *method=None*)

Return an approximation to the minimum Steiner tree of a graph.

The minimum Steiner tree of `G` w.r.t a set of `terminal_nodes` (also *S*) is a tree within `G` that spans those nodes and has minimum size (sum of edge weights) among all such trees.

The approximation algorithm is specified with the `method` keyword argument. All three available algorithms produce a tree whose weight is within a (2 - (2 / l)) factor of the weight of the optimal Steiner tree, where *l* is the minimum number of leaf nodes across all possible Steiner trees.

- `kou` [2] (runtime $O(|S||V|^2)$) computes the minimum spanning tree of

the subgraph of the metric closure of *G* induced by the terminal nodes, where the metric closure of *G* is the complete graph in which each edge is weighted by the shortest path distance between the nodes in *G*.

- `mehlhorn` [3] (runtime $O(|E| + |V| \log |V|)$) modifies Kou et al.'s

algorithm, beginning by finding the closest terminal node for each non-terminal. This data is used to create a complete graph containing only the terminal nodes, in which edge is weighted with the shortest path distance between them. The algorithm then proceeds in the same way as Kou et al..

Parameters

G
[NetworkX graph]

terminal_nodes
[list] A list of terminal nodes for which minimum steiner tree is to be found.

weight
[string (default = 'weight')] Use the edge attribute specified by this string as the edge weight. Any edge attribute not present defaults to 1.

method
[string, optional (default = 'kou')] The algorithm to use to approximate the Steiner tree. Supported options: 'kou', 'mehlhorn'. Other inputs produce a ValueError.

Returns

NetworkX graph
Approximation to the minimum steiner tree of `G` induced by `terminal_nodes` .

Notes

For multigraphs, the edge between two nodes with minimum weight is the edge put into the Steiner tree.

References

[1], [2], [3]

3.1.10 Traveling Salesman

Travelling Salesman Problem (TSP)

Implementation of approximate algorithms for solving and approximating the TSP problem.

Categories of algorithms which are implemented:

- Christofides (provides a 3/2-approximation of TSP)
- Greedy
- Simulated Annealing (SA)
- Threshold Accepting (TA)
- Asadpour Asymmetric Traveling Salesman Algorithm

The Travelling Salesman Problem tries to find, given the weight (distance) between all points where a salesman has to visit, the route so that:

- The total distance (cost) which the salesman travels is minimized.
- The salesman returns to the starting point.
- Note that for a complete graph, the salesman visits each point once.

The function `travelling_salesman_problem` allows for incomplete graphs by finding all-pairs shortest paths, effectively converting the problem to a complete graph problem. It calls one of the approximate methods on that problem and then converts the result back to the original graph using the previously found shortest paths.

TSP is an NP-hard problem in combinatorial optimization, important in operations research and theoretical computer science.

http://en.wikipedia.org/wiki/Travelling_salesman_problem

christofides(G[, weight, tree])	Approximate a solution of the traveling salesman problem
traveling_salesman_problem(G[, weight, ...])	Find the shortest path in `G` connecting specified nodes
greedy_tsp(G[, weight, source])	Return a low cost cycle starting at `source` and its cost.
simulated_annealing_tsp(G, init_cycle[, ...])	Returns an approximate solution to the traveling salesman problem.
threshold_accepting_tsp(G, init_cycle[, ...])	Returns an approximate solution to the traveling salesman problem.
asadpour_atsp(G[, weight, seed, source])	Returns an approximate solution to the traveling salesman problem.

christofides

christofides(*G*, *weight='weight'*, *tree=None*)

Approximate a solution of the traveling salesman problem

Compute a 3/2-approximation of the traveling salesman problem in a complete undirected graph using Christofides [1] algorithm.

Parameters

G
[Graph] `G` should be a complete weighted undirected graph. The distance between all pairs 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.

tree
[NetworkX graph or None (default: None)] A minimum spanning tree of G. Or, if None, the minimum spanning tree is computed using `networkx.minimum_spanning_tree()`

Returns

list
List of nodes in `G` along a cycle with a 3/2-approximation of the minimal Hamiltonian cycle.

References

[1]

traveling_salesman_problem

traveling_salesman_problem(*G*, *weight='weight'*, *nodes=None*, *cycle=True*, *method=None*)

Find the shortest path in `G` connecting specified nodes

This function allows approximate solution to the traveling salesman problem on networks that are not complete graphs and/or where the salesman does not need to visit all nodes.

This function proceeds in two steps. First, it creates a complete graph using the all-pairs shortest_paths between nodes in `nodes`. Edge weights in the new graph are the lengths of the paths between each pair of nodes in the original graph. Second, an algorithm (default: *christofides* for undirected and *asadpour_atsp* for directed) is used to approximate the minimal Hamiltonian cycle on this new graph. The available algorithms are:

- christofides
- greedy_tsp
- simulated_annealing_tsp
- threshold_accepting_tsp
- asadpour_atsp

Once the Hamiltonian Cycle is found, this function post-processes to accommodate the structure of the original graph. If `cycle` is `False`, the biggest weight edge is removed to make a Hamiltonian path. Then each edge on the new complete graph used for that analysis is replaced by the shortest_path between those nodes on the original graph.

Parameters

G
: [NetworkX graph] A possibly weighted graph

nodes
: [collection of nodes (default=G.nodes)] collection (list, set, etc.) of nodes to visit

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.

cycle
: [bool (default: True)] Indicates whether a cycle should be returned, or a path. Note: the cycle is the approximate minimal cycle. The path simply removes the biggest edge in that cycle.

method
: [function (default: None)] A function that returns a cycle on all nodes and approximates the solution to the traveling salesman problem on a complete graph. The returned cycle is then used to find a corresponding solution on `G`. `method` should be callable; take inputs `G`, and `weight`; and return a list of nodes along the cycle.

  Provided options include *christofides()*, *greedy_tsp()*, *simulated_annealing_tsp()* and *threshold_accepting_tsp()*.

  If `method is None`: use *christofides()* for undirected `G` and *threshold_accepting_tsp()* for directed `G`.

  To specify parameters for these provided functions, construct lambda functions that state the specific value. `method` must have 2 inputs. (See examples).

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
```

Build (curry) your own function to provide parameter values to the methods.

```
>>> SA_tsp = nx.approximation.simulated_annealing_tsp
>>> method = lambda G, wt: SA_tsp(G, "greedy", weight=wt, 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
```

greedy_tsp

greedy_tsp(*G*, *weight='weight'*, *source=None*)

Return a low cost cycle starting at `source` and its cost.

This approximates a solution to the traveling salesman problem. It finds a cycle of all the nodes that a salesman can visit in order to visit many nodes while minimizing total distance. It uses a simple greedy algorithm. In essence, this function returns a large cycle given a source point for which the total cost of the cycle is minimized.

Parameters

G
[Graph] The Graph should be a complete weighted undirected graph. The distance between all pairs 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.

source
[node, optional (default: first node in list(G))] Starting node. If None, defaults to `next(iter(G))`

Returns

cycle
[list of nodes] Returns the cycle (list of nodes) that a salesman can follow to minimize total weight of the trip.

Raises

NetworkXError
If `G` is not complete, the algorithm raises an exception.

Notes

This implementation of a greedy algorithm is based on the following:

- The algorithm adds a node to the solution at every iteration.
- The algorithm selects a node not already in the cycle whose connection to the previous node adds the least cost to the cycle.

A greedy algorithm does not always give the best solution. However, it can construct a first feasible solution which can be passed as a parameter to an iterative improvement algorithm such as Simulated Annealing, or Threshold Accepting.

Time complexity: It has a running time $O(|V|^2)$

Examples

```
>>> from networkx.algorithms import approximation as approx
>>> G = nx.DiGraph()
>>> G.add_weighted_edges_from({
...     ("A", "B", 3), ("A", "C", 17), ("A", "D", 14), ("B", "A", 3),
...     ("B", "C", 12), ("B", "D", 16), ("C", "A", 13),("C", "B", 12),
...     ("C", "D", 4), ("D", "A", 14), ("D", "B", 15), ("D", "C", 2)
... })
>>> cycle = approx.greedy_tsp(G, source="D")
>>> cost = sum(G[n][nbr]["weight"] for n, nbr in nx.utils.pairwise(cycle))
>>> cycle
['D', 'C', 'B', 'A', 'D']
>>> cost
31
```

simulated_annealing_tsp

simulated_annealing_tsp(*G*, *init_cycle*, *weight='weight'*, *source=None*, *temp=100*, *move='1-1'*, *max_iterations=10*, *N_inner=100*, *alpha=0.01*, *seed=None*)

Returns an approximate solution to the traveling salesman problem.

This function uses simulated annealing to approximate the minimal cost cycle through the nodes. Starting from a suboptimal solution, simulated annealing perturbs that solution, occasionally accepting changes that make the solution worse to escape from a locally optimal solution. The chance of accepting such changes decreases over the iterations to encourage an optimal result. In summary, the function returns a cycle starting at `source` for which the total cost is minimized. It also returns the cost.

The chance of accepting a proposed change is related to a parameter called the temperature (annealing has a physical analogue of steel hardening as it cools). As the temperature is reduced, the chance of moves that increase cost goes down.

Parameters

G
: [Graph] `G` should be a complete weighted undirected graph. The distance between all pairs of nodes should be included.

init_cycle
: [list of all nodes or "greedy"] The initial solution (a cycle through all nodes returning to the start). This argument has no default to make you think about it. If "greedy", use `greedy_tsp(G, weight)`. Other common starting cycles are `list(G) + [next(iter(G))]` or the final result of *simulated_annealing_tsp* when doing *threshold_accepting_tsp*.

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.

source
: [node, optional (default: first node in list(G))] Starting node. If None, defaults to `next(iter(G))`

temp
: [int, optional (default=100)] The algorithm's temperature parameter. It represents the initial value of temperature

move
["1-1" or "1-0" or function, optional (default="1-1")] Indicator of what move to use when finding new trial solutions. Strings indicate two special built-in moves:

- "1-1": 1-1 exchange which transposes the position of two elements of the current solution. The function called is `swap_two_nodes()`. For example if we apply 1-1 exchange in the solution `A = [3, 2, 1, 4, 3]` we can get the following by the transposition of 1 and 4 elements: `A' = [3, 2, 4, 1, 3]`
- "1-0": 1-0 exchange which moves an node in the solution to a new position. The function called is `move_one_node()`. For example if we apply 1-0 exchange in the solution `A = [3, 2, 1, 4, 3]` we can transfer the fourth element to the second position: `A' = [3, 4, 2, 1, 3]`

You may provide your own functions to enact a move from one solution to a neighbor solution. The function must take the solution as input along with a `seed` input to control random number generation (see the `seed` input here). Your function should maintain the solution as a cycle with equal first and last node and all others appearing once. Your function should return the new solution.

max_iterations
[int, optional (default=10)] Declared done when this number of consecutive iterations of the outer loop occurs without any change in the best cost solution.

N_inner
[int, optional (default=100)] The number of iterations of the inner loop.

alpha
[float between (0, 1), optional (default=0.01)] Percentage of temperature decrease in each iteration of outer loop

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

cycle
[list of nodes] Returns the cycle (list of nodes) that a salesman can follow to minimize total weight of the trip.

Raises

NetworkXError
If `G` is not complete the algorithm raises an exception.

Notes

Simulated Annealing is a metaheuristic local search algorithm. The main characteristic of this algorithm is that it accepts even solutions which lead to the increase of the cost in order to escape from low quality local optimal solutions.

This algorithm needs an initial solution. If not provided, it is constructed by a simple greedy algorithm. At every iteration, the algorithm selects thoughtfully a neighbor solution. Consider $c(x)$ cost of current solution and $c(x')$ cost of a neighbor solution. If $c(x') - c(x) <= 0$ then the neighbor solution becomes the current solution for the next iteration. Otherwise, the algorithm accepts the neighbor solution with probability $p = exp - ([c(x') - c(x)]/temp)$. Otherwise the current solution is retained.

`temp` is a parameter of the algorithm and represents temperature.

Time complexity: For N_i iterations of the inner loop and N_o iterations of the outer loop, this algorithm has running time $O(N_i * N_o * |V|)$.

For more information and how the algorithm is inspired see: http://en.wikipedia.org/wiki/Simulated_annealing

Examples

```
>>> from networkx.algorithms import approximation as approx
>>> G = nx.DiGraph()
>>> G.add_weighted_edges_from({
...     ("A", "B", 3), ("A", "C", 17), ("A", "D", 14), ("B", "A", 3),
...     ("B", "C", 12), ("B", "D", 16), ("C", "A", 13),("C", "B", 12),
...     ("C", "D", 4), ("D", "A", 14), ("D", "B", 15), ("D", "C", 2)
... })
>>> cycle = approx.simulated_annealing_tsp(G, "greedy", source="D")
>>> cost = sum(G[n][nbr]["weight"] for n, nbr in nx.utils.pairwise(cycle))
>>> cycle
['D', 'C', 'B', 'A', 'D']
>>> cost
31
>>> incycle = ["D", "B", "A", "C", "D"]
>>> cycle = approx.simulated_annealing_tsp(G, incycle, source="D")
>>> cost = sum(G[n][nbr]["weight"] for n, nbr in nx.utils.pairwise(cycle))
>>> cycle
['D', 'C', 'B', 'A', 'D']
>>> cost
31
```

threshold_accepting_tsp

threshold_accepting_tsp(*G*, *init_cycle*, *weight='weight'*, *source=None*, *threshold=1*, *move='1-1'*, *max_iterations=10*, *N_inner=100*, *alpha=0.1*, *seed=None*)

Returns an approximate solution to the traveling salesman problem.

This function uses threshold accepting methods to approximate the minimal cost cycle through the nodes. Starting from a suboptimal solution, threshold accepting methods perturb that solution, accepting any changes that make the solution no worse than increasing by a threshold amount. Improvements in cost are accepted, but so are changes leading to small increases in cost. This allows the solution to leave suboptimal local minima in solution space. The threshold is decreased slowly as iterations proceed helping to ensure an optimum. In summary, the function returns a cycle starting at `source` for which the total cost is minimized.

Parameters

G

[Graph] `G` should be a complete weighted undirected graph. The distance between all pairs of nodes should be included.

init_cycle

[list or "greedy"] The initial solution (a cycle through all nodes returning to the start). This argument has no default to make you think about it. If "greedy", use `greedy_tsp(G, weight)`. Other common starting cycles are `list(G) + [next(iter(G))]` or the final result of *simulated_annealing_tsp* when doing *threshold_accepting_tsp*.

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.

source
[node, optional (default: first node in list(G))] Starting node. If None, defaults to `next(iter(G))`

threshold
[int, optional (default=1)] The algorithm's threshold parameter. It represents the initial threshold's value

move
["1-1" or "1-0" or function, optional (default="1-1")] Indicator of what move to use when finding new trial solutions. Strings indicate two special built-in moves:

- "1-1": 1-1 exchange which transposes the position of two elements of the current solution. The function called is `swap_two_nodes()`. For example if we apply 1-1 exchange in the solution `A = [3, 2, 1, 4, 3]` we can get the following by the transposition of 1 and 4 elements: `A' = [3, 2, 4, 1, 3]`
- "1-0": 1-0 exchange which moves an node in the solution to a new position. The function called is `move_one_node()`. For example if we apply 1-0 exchange in the solution `A = [3, 2, 1, 4, 3]` we can transfer the fourth element to the second position: `A' = [3, 4, 2, 1, 3]`

You may provide your own functions to enact a move from one solution to a neighbor solution. The function must take the solution as input along with a `seed` input to control random number generation (see the `seed` input here). Your function should maintain the solution as a cycle with equal first and last node and all others appearing once. Your function should return the new solution.

max_iterations
[int, optional (default=10)] Declared done when this number of consecutive iterations of the outer loop occurs without any change in the best cost solution.

N_inner
[int, optional (default=100)] The number of iterations of the inner loop.

alpha
[float between (0, 1), optional (default=0.1)] Percentage of threshold decrease when there is at least one acceptance of a neighbor solution. If no inner loop moves are accepted the threshold remains unchanged.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

cycle
[list of nodes] Returns the cycle (list of nodes) that a salesman can follow to minimize total weight of the trip.

Raises

NetworkXError
If `G` is not complete the algorithm raises an exception.

See also:

simulated_annealing_tsp

Notes

Threshold Accepting is a metaheuristic local search algorithm. The main characteristic of this algorithm is that it accepts even solutions which lead to the increase of the cost in order to escape from low quality local optimal solutions.

This algorithm needs an initial solution. This solution can be constructed by a simple greedy algorithm. At every iteration, it selects thoughtfully a neighbor solution. Consider $c(x)$ cost of current solution and $c(x')$ cost of neighbor solution. If $c(x') - c(x) <= threshold$ then the neighbor solution becomes the current solution for the next iteration, where the threshold is named threshold.

In comparison to the Simulated Annealing algorithm, the Threshold Accepting algorithm does not accept very low quality solutions (due to the presence of the threshold value). In the case of Simulated Annealing, even a very low quality solution can be accepted with probability p.

Time complexity: It has a running time $O(m * n * |V|)$ where m and n are the number of times the outer and inner loop run respectively.

For more information and how algorithm is inspired see: https://doi.org/10.1016/0021-9991(90)90201-B

Examples

```
>>> from networkx.algorithms import approximation as approx
>>> G = nx.DiGraph()
>>> G.add_weighted_edges_from({
...     ("A", "B", 3), ("A", "C", 17), ("A", "D", 14), ("B", "A", 3),
...     ("B", "C", 12), ("B", "D", 16), ("C", "A", 13),("C", "B", 12),
...     ("C", "D", 4), ("D", "A", 14), ("D", "B", 15), ("D", "C", 2)
... })
>>> cycle = approx.threshold_accepting_tsp(G, "greedy", source="D")
>>> cost = sum(G[n][nbr]["weight"] for n, nbr in nx.utils.pairwise(cycle))
>>> cycle
['D', 'C', 'B', 'A', 'D']
>>> cost
31
>>> incycle = ["D", "B", "A", "C", "D"]
>>> cycle = approx.threshold_accepting_tsp(G, incycle, source="D")
>>> cost = sum(G[n][nbr]["weight"] for n, nbr in nx.utils.pairwise(cycle))
>>> cycle
['D', 'C', 'B', 'A', 'D']
>>> cost
31
```

asadpour_atsp

asadpour_atsp(*G*, *weight='weight'*, *seed=None*, *source=None*)

Returns an approximate solution to the traveling salesman problem.

This approximate solution is one of the best known approximations for the asymmetric traveling salesman problem developed by Asadpour et al, [1]. The algorithm first solves the Held-Karp relaxation to find a lower bound for the weight of the cycle. Next, it constructs an exponential distribution of undirected spanning trees where the probability of an edge being in the tree corresponds to the weight of that edge using a maximum entropy rounding scheme. Next we sample that distribution $2\lceil \ln n \rceil$ times and save the minimum sampled tree once the direction of the arcs is added back to the edges. Finally, we augment then short circuit that graph to find the approximate tour for the salesman.

Parameters

G
[nx.DiGraph] The graph should be a complete weighted directed graph. The distance between all paris of nodes should be included and the triangle inequality should hold. That is, the direct edge between any two nodes should be the path of least cost.

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.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

source
[node label (default=`None`)] If given, return the cycle starting and ending at the given node.

Returns

cycle
[list of nodes] Returns the cycle (list of nodes) that a salesman can follow to minimize the total weight of the trip.

Raises

NetworkXError
If `G` is not complete or has less than two nodes, the algorithm raises an exception.

NetworkXError
If `source` is not `None` and is not a node in `G`, the algorithm raises an exception.

NetworkXNotImplemented
If `G` is an undirected graph.

References

[1]

Examples

```
>>> import networkx as nx
>>> import networkx.algorithms.approximation as approx
>>> G = nx.complete_graph(3, create_using=nx.DiGraph)
>>> nx.set_edge_attributes(G, {(0, 1): 2, (1, 2): 2, (2, 0): 2, (0, 2): 1, (2,␣
↪1): 1, (1, 0): 1}, "weight")
>>> tour = approx.asadpour_atsp(G,source=0)
>>> tour
[0, 2, 1, 0]
```

3.1.11 Treewidth

Functions for computing treewidth decomposition.

Treewidth of an undirected graph is a number associated with the graph. It can be defined as the size of the largest vertex set (bag) in a tree decomposition of the graph minus one.

Wikipedia: Treewidth

The notions of treewidth and tree decomposition have gained their attractiveness partly because many graph and network problems that are intractable (e.g., NP-hard) on arbitrary graphs become efficiently solvable (e.g., with a linear time algorithm) when the treewidth of the input graphs is bounded by a constant [1] [2].

There are two different functions for computing a tree decomposition: *treewidth_min_degree()* and *treewidth_min_fill_in()*.

treewidth_min_degree(G)	Returns a treewidth decomposition using the Minimum Degree heuristic.
treewidth_min_fill_in(G)	Returns a treewidth decomposition using the Minimum Fill-in heuristic.

treewidth_min_degree

treewidth_min_degree(*G*)

Returns a treewidth decomposition using the Minimum Degree heuristic.

The heuristic chooses the nodes according to their degree, i.e., first the node with the lowest degree is chosen, then the graph is updated and the corresponding node is removed. Next, a new node with the lowest degree is chosen, and so on.

Parameters

G
[NetworkX graph]

Returns

Treewidth decomposition
[(int, Graph) tuple] 2-tuple with treewidth and the corresponding decomposed tree.

treewidth_min_fill_in

treewidth_min_fill_in(*G*)

Returns a treewidth decomposition using the Minimum Fill-in heuristic.

The heuristic chooses a node from the graph, where the number of edges added turning the neighbourhood of the chosen node into clique is as small as possible.

Parameters

G
[NetworkX graph]

Returns

Treewidth decomposition
[(int, Graph) tuple] 2-tuple with treewidth and the corresponding decomposed tree.

3.1.12 Vertex Cover

Functions for computing an approximate minimum weight vertex cover.

A *vertex cover* is a subset of nodes such that each edge in the graph is incident to at least one node in the subset.

min_weighted_vertex_cover(G[, weight])	Returns an approximate minimum weighted vertex cover.

min_weighted_vertex_cover

min_weighted_vertex_cover(*G*, *weight=None*)

Returns an approximate minimum weighted vertex cover.

The set of nodes returned by this function is guaranteed to be a vertex cover, and the total weight of the set is guaranteed to be at most twice the total weight of the minimum weight vertex cover. In other words,

$$w(S) \leq 2 * w(S^*),$$

where S is the vertex cover returned by this function, S^* is the vertex cover of minimum weight out of all vertex covers of the graph, and w is the function that computes the sum of the weights of each node in that given set.

Parameters

G
[NetworkX graph]

weight
[string, optional (default = None)] If None, every node has weight 1. If a string, use this node attribute as the node weight. A node without this attribute is assumed to have weight 1.

Returns

min_weighted_cover
[set] Returns a set of nodes whose weight sum is no more than twice the weight sum of the minimum weight vertex cover.

Notes

For a directed graph, a vertex cover has the same definition: a set of nodes such that each edge in the graph is incident to at least one node in the set. Whether the node is the head or tail of the directed edge is ignored.

This is the local-ratio algorithm for computing an approximate vertex cover. The algorithm greedily reduces the costs over edges, iteratively building a cover. The worst-case runtime of this implementation is $O(m \log n)$, where n is the number of nodes and m the number of edges in the graph.

References

[1]

3.1.13 Max Cut

randomized_partitioning(G[, seed, p, weight])	Compute a random partitioning of the graph nodes and its cut value.
one_exchange(G[, initial_cut, seed, weight])	Compute a partitioning of the graphs nodes and the corresponding cut value.

randomized_partitioning

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 *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.

one_exchange

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 *Randomness*.

weight
[object] Edge attribute key to use as weight. If not specified, edges have weight one.

Returns

cut_value
[scalar] Value of the maximum cut.

partition
[pair of node sets] A partitioning of the nodes that defines a maximum cut.

3.2 Assortativity

3.2.1 Assortativity

degree_assortativity_coefficient(G[, x, y, ...])	Compute degree assortativity of graph.
attribute_assortativity_coefficient(G, attribute)	Compute assortativity for node attributes.
numeric_assortativity_coefficient(G, attribute)	Compute assortativity for numerical node attributes.
degree_pearson_correlation_coefficient(G[, ...])	Compute degree assortativity of graph.

degree_assortativity_coefficient

degree_assortativity_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.

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 degree assortativity only for nodes in container. The default is all nodes.

Returns

r
[float] Assortativity of graph by degree.

See also:

attribute_assortativity_coefficient
numeric_assortativity_coefficient
degree_mixing_dict
degree_mixing_matrix

Notes

This computes Eq. (21) in Ref. [1] , where e is the joint probability distribution (mixing matrix) of the degrees. If G is directed than the matrix e is the joint probability of the user-specified degree type for the source and target.

References

[1], [2]

Examples

```
>>> G = nx.path_graph(4)
>>> r = nx.degree_assortativity_coefficient(G)
>>> print(f"{r:3.1f}")
-0.5
```

attribute_assortativity_coefficient

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

Notes

This computes Eq. (2) in Ref. [1] , (trace(M)-sum(M^2))/(1-sum(M^2)), where M is the joint probability distribution (mixing matrix) of the specified attribute.

References

[1]

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"))
1.0
```

numeric_assortativity_coefficient

numeric_assortativity_coefficient(*G*, *attribute*, *nodes=None*)

Compute assortativity for numerical node attributes.

Assortativity measures the similarity of connections in the graph with respect to the given numeric attribute.

Parameters

G
[NetworkX graph]

attribute
[string] Node attribute key.

nodes: list or iterable (optional)
Compute numeric assortativity only for attributes of nodes in container. The default is all nodes.

Returns

r: float
Assortativity of graph for given attribute

Notes

This computes Eq. (21) in Ref. [1] , which is the Pearson correlation coefficient of the specified (scalar valued) attribute across edges.

References

[1]

Examples

```
>>> G = nx.Graph()
>>> G.add_nodes_from([0, 1], size=2)
>>> G.add_nodes_from([2, 3], size=3)
>>> G.add_edges_from([(0, 1), (2, 3)])
>>> print(nx.numeric_assortativity_coefficient(G, "size"))
1.0
```

degree_pearson_correlation_coefficient

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.

Notes

This calls scipy.stats.pearsonr.

References

[1], [2]

Examples

```
>>> G = nx.path_graph(4)
>>> r = nx.degree_pearson_correlation_coefficient(G)
>>> print(f"{r:3.1f}")
-0.5
```

3.2.2 Average neighbor degree

average_neighbor_degree(G[, source, target, ...])	Returns the average degree of the neighborhood of each node.

average_neighbor_degree

average_neighbor_degree(*G*, *source='out'*, *target='out'*, *nodes=None*, *weight=None*)

Returns the average degree of the neighborhood of each node.

In an undirected graph, the neighborhood `N(i)` of node `i` contains the nodes that are connected to `i` by an edge.

For directed graphs, `N(i)` is defined according to the parameter `source`:

- if source is 'in', then `N(i)` consists of predecessors of node `i`.
- if source is 'out', then `N(i)` consists of successors of node `i`.
- if source is 'in+out', then `N(i)` is both predecessors and successors.

The average neighborhood degree of a node `i` is

$$k_{nn,i} = \frac{1}{|N(i)|} \sum_{j \in N(i)} k_j$$

where `N(i)` are the neighbors of node `i` and `k_j` is the degree of node `j` which belongs to `N(i)`. For weighted graphs, an analogous measure can be defined [1],

$$k^w_{nn,i} = \frac{1}{s_i} \sum_{j \in N(i)} w_{ij} k_j$$

where `s_i` is the weighted degree of node `i`, `w_{ij}` is the weight of the edge that links `i` and `j` and `N(i)` are the neighbors of node `i`.

Parameters

G
[NetworkX graph]

source
[string ("in"|"out"|"in+out"), optional (default="out")] Directed graphs only. Use "in"- or "out"-neighbors of source node.

target
[string ("in"|"out"|"in+out"), optional (default="out")] Directed graphs only. Use "in"- or "out"-degree for target node.

nodes
[list or iterable, optional (default=G.nodes)] Compute neighbor degree only for specified nodes.

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

d: dict
A dictionary keyed by node to the average degree of its neighbors.

Raises

NetworkXError
If either `source` or `target` are not one of 'in', 'out', or 'in+out'. If either `source` or `target` is passed for an undirected graph.

See also:

average_degree_connectivity

References

[1]

Examples

```
>>> G = nx.path_graph(4)
>>> G.edges[0, 1]["weight"] = 5
>>> G.edges[2, 3]["weight"] = 3
```

```
>>> nx.average_neighbor_degree(G)
{0: 2.0, 1: 1.5, 2: 1.5, 3: 2.0}
>>> nx.average_neighbor_degree(G, weight="weight")
{0: 2.0, 1: 1.1666666666666667, 2: 1.25, 3: 2.0}
```

```
>>> G = nx.DiGraph()
>>> nx.add_path(G, [0, 1, 2, 3])
>>> nx.average_neighbor_degree(G, source="in", target="in")
{0: 0.0, 1: 0.0, 2: 1.0, 3: 1.0}
```

```
>>> nx.average_neighbor_degree(G, source="out", target="out")
{0: 1.0, 1: 1.0, 2: 0.0, 3: 0.0}
```

3.2.3 Average degree connectivity

average_degree_connectivity(G[, ...])	Compute the average degree connectivity of graph.

average_degree_connectivity

average_degree_connectivity(*G*, *source='in+out'*, *target='in+out'*, *nodes=None*, *weight=None*)

Compute the average degree connectivity of graph.

The average degree connectivity is the average nearest neighbor degree of nodes with degree k. For weighted graphs, an analogous measure can be computed using the weighted average neighbors degree defined in [1], for a node `i`, as

$$k_{nn,i}^{w} = \frac{1}{s_i} \sum_{j \in N(i)} w_{ij} k_j$$

where `s_i` is the weighted degree of node `i`, `w_{ij}` is the weight of the edge that links `i` and `j`, and `N(i)` are the neighbors of node `i`.

Parameters

G
[NetworkX graph]

source
["in"|"out"|"in+out" (default:"in+out")] Directed graphs only. Use "in"- or "out"-degree for source node.

target
["in"|"out"|"in+out" (default:"in+out"] Directed graphs only. Use "in"- or "out"-degree for target node.

nodes
[list or iterable (optional)] Compute neighbor connectivity for these nodes. The default is all nodes.

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

d
[dict] A dictionary keyed by degree k with the value of average connectivity.

Raises

NetworkXError
If either `source` or `target` are not one of 'in', 'out', or 'in+out'. If either `source` or `target` is passed for an undirected graph.

See also:

average_neighbor_degree

References

[1]

Examples

```
>>> G = nx.path_graph(4)
>>> G.edges[1, 2]["weight"] = 3
>>> nx.average_degree_connectivity(G)
{1: 2.0, 2: 1.5}
>>> nx.average_degree_connectivity(G, weight="weight")
{1: 2.0, 2: 1.75}
```

3.2.4 Mixing

attribute_mixing_matrix(G, attribute[, ...])	Returns mixing matrix for attribute.
degree_mixing_matrix(G[, x, y, weight, ...])	Returns mixing matrix for attribute.
attribute_mixing_dict(G, attribute[, nodes, ...])	Returns dictionary representation of mixing matrix for attribute.
degree_mixing_dict(G[, x, y, weight, nodes, ...])	Returns dictionary representation of mixing matrix for degree.
mixing_dict(xy[, normalized])	Returns a dictionary representation of mixing matrix.

attribute_mixing_matrix

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)
>>> # mixing from male nodes to female nodes
>>> mix_mat[mapping['male'], mapping['female']]
0.25
```

degree_mixing_matrix

degree_mixing_matrix(*G*, *x='out'*, *y='in'*, *weight=None*, *nodes=None*, *normalized=True*, *mapping=None*)

Returns mixing matrix for attribute.

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).

nodes: list or iterable (optional)
Build the matrix using only nodes in container. The default is all nodes.

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=True)] Return counts if False or probabilities if True.

mapping
[dictionary, optional] Mapping from node degree to integer index in matrix. If not specified, an arbitrary ordering will be used.

Returns

m: numpy array
Counts, or joint probability, of occurrence of node degree.

Notes

Definitions of degree mixing matrix vary on whether the matrix should include rows for degree 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. See examples.

Examples

```
>>> G = nx.star_graph(3)
>>> mix_mat = nx.degree_mixing_matrix(G)
>>> mix_mat[0, 1]  # mixing from node degree 1 to node degree 3
0.5
```

If you want every possible degree to appear as a row, even if no nodes have that degree, use `mapping` as follows,

```
>>> max_degree = max(deg for n, deg in G.degree)
>>> mapping = {x: x for x in range(max_degree + 1)} # identity mapping
>>> mix_mat = nx.degree_mixing_matrix(G, mapping=mapping)
>>> mix_mat[3, 1]  # mixing from node degree 3 to node degree 1
0.5
```

attribute_mixing_dict

attribute_mixing_dict(*G*, *attribute*, *nodes=None*, *normalized=False*)

Returns dictionary representation of mixing matrix for attribute.

Parameters

G
[graph] NetworkX graph object.

attribute
[string] Node attribute key.

nodes: list or iterable (optional)
Unse nodes in container to build the dict. The default is all nodes.

normalized
[bool (default=False)] Return counts if False or probabilities if True.

Returns

d
[dictionary] Counts or joint probability of occurrence of attribute pairs.

Examples

```
>>> G = nx.Graph()
>>> G.add_nodes_from([0, 1], color="red")
>>> G.add_nodes_from([2, 3], color="blue")
>>> G.add_edge(1, 3)
>>> d = nx.attribute_mixing_dict(G, "color")
>>> print(d["red"]["blue"])
1
```

(continues on next page)

(continued from previous page)

```
>>> print(d["blue"]["red"])  # d symmetric for undirected graphs
1
```

degree_mixing_dict

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)] Return counts if False or probabilities if True.

Returns

d: dictionary
Counts or joint probability of occurrence of degree pairs.

mixing_dict

mixing_dict(*xy*, *normalized=False*)

Returns a dictionary representation of mixing matrix.

Parameters

xy
[list or container of two-tuples] Pairs of (x,y) items.

attribute
[string] Node attribute key

normalized
[bool (default=False)] Return counts if False or probabilities if True.

Returns

d: dictionary
Counts or Joint probability of occurrence of values in xy.

3.2.5 Pairs

node_attribute_xy(G, attribute[, nodes])	Returns iterator of node-attribute pairs for all edges in G.
node_degree_xy(G[, x, y, weight, nodes])	Generate node degree-degree pairs for edges in G.

node_attribute_xy

node_attribute_xy(*G*, *attribute*, *nodes=None*)

Returns iterator of node-attribute pairs for all edges in G.

Parameters

G: NetworkX graph
attribute: key
The node attribute key.

nodes: list or iterable (optional)
Use only edges that are incident to specified nodes. The default is all nodes.

Returns

(x, y): 2-tuple
Generates 2-tuple of (attribute, attribute) values.

Notes

For undirected graphs each edge is produced twice, once for each edge representation (u, v) and (v, u), with the exception of self-loop edges which only appear once.

Examples

```
>>> G = nx.DiGraph()
>>> G.add_node(1, color="red")
>>> G.add_node(2, color="blue")
>>> G.add_edge(1, 2)
>>> list(nx.node_attribute_xy(G, "color"))
[('red', 'blue')]
```

node_degree_xy

node_degree_xy(*G*, *x='out'*, *y='in'*, *weight=None*, *nodes=None*)

Generate node degree-degree pairs for edges in G.

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)
Use only edges that are adjacency to specified nodes. The default is all nodes.

Returns

(x, y): 2-tuple
Generates 2-tuple of (degree, degree) values.

Notes

For undirected graphs each edge is produced twice, once for each edge representation (u, v) and (v, u), with the exception of self-loop edges which only appear once.

Examples

```
>>> G = nx.DiGraph()
>>> G.add_edge(1, 2)
>>> list(nx.node_degree_xy(G, x="out", y="in"))
[(1, 1)]
>>> list(nx.node_degree_xy(G, x="in", y="out"))
[(0, 0)]
```

3.3 Asteroidal

Algorithms for asteroidal triples and asteroidal numbers in graphs.

An asteroidal triple in a graph G is a set of three non-adjacent vertices u, v and w such that there exist a path between any two of them that avoids closed neighborhood of the third. More formally, v_j, v_k belongs to the same connected component of G - N[v_i], where N[v_i] denotes the closed neighborhood of v_i. A graph which does not contain any asteroidal triples is called an AT-free graph. The class of AT-free graphs is a graph class for which many NP-complete problems are solvable in polynomial time. Amongst them, independent set and coloring.

is_at_free(G)	Check if a graph is AT-free.
find_asteroidal_triple(G)	Find an asteroidal triple in the given graph.

3.3.1 is_at_free

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
```

3.3.2 find_asteroidal_triple

find_asteroidal_triple(*G*)

Find an asteroidal triple in the given graph.

An asteroidal triple is a triple of non-adjacent vertices such that there exists a path between any two of them which avoids the closed neighborhood of the third. It checks all independent triples of vertices and whether they are an asteroidal triple or not. This is done with the help of a data structure called a component structure. A component structure encodes information about which vertices belongs to the same connected component when the closed neighborhood of a given vertex is removed from the graph. The algorithm used to check is the trivial one, outlined in [1], which has a runtime of $O(|V||\overline{E}+|V||E|)$, where the second term is the creation of the component structure.

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]

3.4 Bipartite

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 *AmbiguousSolution* Exception if the input graph for *bipartite.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 *Generators*.

3.4.1 Basic functions

is_bipartite(G)	Returns True if graph G is bipartite, False if not.
is_bipartite_node_set(G, nodes)	Returns True if nodes and G/nodes are a bipartition of G.
sets(G[, top_nodes])	Returns bipartite node sets of graph G.
color(G)	Returns a two-coloring of the graph.
density(B, nodes)	Returns density of bipartite graph B.
degrees(B, nodes[, weight])	Returns the degrees of the two node sets in the bipartite graph B.

is_bipartite

is_bipartite(*G*)

Returns True if graph G is bipartite, False if not.

Parameters

G

[NetworkX graph]

See also:

color, ***is_bipartite_node_set***

Examples

```
>>> from networkx.algorithms import bipartite
>>> G = nx.path_graph(4)
>>> print(bipartite.is_bipartite(G))
True
```

is_bipartite_node_set

is_bipartite_node_set(*G*, *nodes*)

Returns True if nodes and G/nodes are a bipartition of G.

Parameters

G
[NetworkX graph]

nodes: list or container
Check if nodes are a one of a bipartite set.

Notes

An exception is raised if the input nodes are not distinct, because in this case some bipartite algorithms will yield incorrect results. For connected graphs the bipartite sets are unique. This function handles disconnected graphs.

Examples

```
>>> from networkx.algorithms import bipartite
>>> G = nx.path_graph(4)
>>> X = set([1, 3])
>>> bipartite.is_bipartite_node_set(G, X)
True
```

sets

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 *bipartite documentation* 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.

See also:

color

Examples

```
>>> from networkx.algorithms import bipartite
>>> G = nx.path_graph(4)
>>> X, Y = bipartite.sets(G)
>>> list(X)
[0, 2]
>>> list(Y)
[1, 3]
```

color

color(*G*)

Returns a two-coloring of the graph.

Raises an exception if the graph is not bipartite.

Parameters

G
[NetworkX graph]

Returns

color
[dictionary] A dictionary keyed by node with a 1 or 0 as data for each node color.

Raises

NetworkXError
If the graph is not two-colorable.

Examples

```
>>> from networkx.algorithms import bipartite
>>> G = nx.path_graph(4)
>>> c = bipartite.color(G)
>>> print(c)
{0: 1, 1: 0, 2: 1, 3: 0}
```

You can use this to set a node attribute indicating the biparite set:

```
>>> nx.set_node_attributes(G, c, "bipartite")
>>> print(G.nodes[0]["bipartite"])
1
>>> print(G.nodes[1]["bipartite"])
0
```

density

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.

Returns

d
[float] The bipartite density

See also:

color

Notes

The container of nodes passed as argument must contain all nodes in one of the two bipartite node sets to avoid ambiguity in the case of disconnected graphs. See *bipartite documentation* for further details on how bipartite graphs are handled in NetworkX.

Examples

```
>>> from networkx.algorithms import bipartite
>>> G = nx.complete_bipartite_graph(3, 2)
>>> X = set([0, 1, 2])
>>> bipartite.density(G, X)
1.0
>>> Y = set([3, 4])
>>> bipartite.density(G, Y)
1.0
```

degrees

degrees(*B*, *nodes*, *weight=None*)

Returns the degrees of the two node sets in the bipartite graph B.

Parameters

B
[NetworkX graph]

nodes: list or container
Nodes in one node set of the bipartite graph.

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.

Returns

(degX,degY)
[tuple of dictionaries] The degrees of the two bipartite sets as dictionaries keyed by node.

See also:

color, *density*

Notes

The container of nodes passed as argument must contain all nodes in one of the two bipartite node sets to avoid ambiguity in the case of disconnected graphs. See *bipartite documentation* for further details on how bipartite graphs are handled in NetworkX.

Examples

```
>>> from networkx.algorithms import bipartite
>>> G = nx.complete_bipartite_graph(3, 2)
>>> Y = set([3, 4])
>>> degX, degY = bipartite.degrees(G, Y)
>>> dict(degX)
{0: 2, 1: 2, 2: 2}
```

3.4.2 Edgelist

Read and write NetworkX graphs as bipartite edge lists.

Format

You can read or write three formats of edge lists with these functions.

Node pairs with no data:

```
1 2
```

Python dictionary as data:

```
1 2 {'weight':7, 'color':'green'}
```

Arbitrary data:

```
1 2 7 green
```

For each edge (u, v) the node u is assigned to part 0 and the node v to part 1.

generate_edgelist(G[, delimiter, data])	Generate a single line of the bipartite graph G in edge list format.
write_edgelist(G, path[, comments, ...])	Write a bipartite graph as a list of edges.
parse_edgelist(lines[, comments, delimiter, ...])	Parse lines of an edge list representation of a bipartite graph.
read_edgelist(path[, comments, delimiter, ...])	Read a bipartite graph from a list of edges.

generate_edgelist

generate_edgelist(*G*, *delimiter=' '*, *data=True*)

Generate a single line of the bipartite graph G in edge list format.

Parameters

G
[NetworkX graph] The graph is assumed to have node attribute `part` set to 0,1 representing the two graph parts

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

```
>>> from networkx.algorithms import bipartite
>>> G = nx.path_graph(4)
>>> G.add_nodes_from([0, 2], bipartite=0)
>>> G.add_nodes_from([1, 3], bipartite=1)
>>> G[1][2]["weight"] = 3
>>> G[2][3]["capacity"] = 12
>>> for line in bipartite.generate_edgelist(G, data=False):
...     print(line)
0 1
2 1
2 3
```

```
>>> for line in bipartite.generate_edgelist(G):
...     print(line)
0 1 {}
2 1 {'weight': 3}
2 3 {'capacity': 12}
```

```
>>> for line in bipartite.generate_edgelist(G, data=["weight"]):
...     print(line)
0 1
2 1 3
2 3
```

write_edgelist

write_edgelist(*G*, *path*, *comments='#'*, *delimiter=' '*, *data=True*, *encoding='utf-8'*)

Write a bipartite graph as a list of edges.

Parameters

G
: [Graph] A NetworkX bipartite graph

path
: [file or string] File or filename to write. If a file is provided, it must be opened in 'wb' mode. Filenames ending in .gz or .bz2 will be compressed.

comments
: [string, optional] The character used to indicate the start of a comment

delimiter
: [string, optional] The string used to separate values. The default is whitespace.

data
: [bool or list, optional] If False write no edge data. If True write a string representation of the edge data dictionary.. If a list (or other iterable) is provided, write the keys specified in the list.

encoding: string, optional
: Specify which encoding to use when writing file.

See also:

write_edgelist
generate_edgelist

Examples

```
>>> G = nx.path_graph(4)
>>> G.add_nodes_from([0, 2], bipartite=0)
>>> G.add_nodes_from([1, 3], bipartite=1)
>>> nx.write_edgelist(G, "test.edgelist")
>>> fh = open("test.edgelist", "wb")
>>> nx.write_edgelist(G, fh)
>>> nx.write_edgelist(G, "test.edgelist.gz")
>>> nx.write_edgelist(G, "test.edgelist.gz", data=False)
```

```
>>> G = nx.Graph()
>>> G.add_edge(1, 2, weight=7, color="red")
>>> nx.write_edgelist(G, "test.edgelist", data=False)
>>> nx.write_edgelist(G, "test.edgelist", data=["color"])
>>> nx.write_edgelist(G, "test.edgelist", data=["color", "weight"])
```

parse_edgelist

parse_edgelist(*lines*, *comments='#'*, *delimiter=None*, *create_using=None*, *nodetype=None*, *data=True*)

Parse lines of an edge list representation of a bipartite graph.

Parameters

lines
[list or iterator of strings] Input data in edgelist format

comments
[string, optional] Marker for comment lines

delimiter
[string, optional] Separator for node labels

create_using: NetworkX graph container, optional
Use given NetworkX graph for holding nodes or edges.

nodetype
[Python type, optional] Convert nodes to this type.

data
[bool or list of (label,type) tuples] If False generate no edge data or if True use a dictionary representation of edge data or a list tuples specifying dictionary key names and types for edge data.

Returns

G: NetworkX Graph
The bipartite graph corresponding to lines

Examples

Edgelist with no data:

```
>>> from networkx.algorithms import bipartite
>>> lines = ["1 2", "2 3", "3 4"]
>>> G = bipartite.parse_edgelist(lines, nodetype=int)
>>> sorted(G.nodes())
[1, 2, 3, 4]
>>> sorted(G.nodes(data=True))
[(1, {'bipartite': 0}), (2, {'bipartite': 0}), (3, {'bipartite': 0}), (4, {
↪'bipartite': 1})]
>>> sorted(G.edges())
[(1, 2), (2, 3), (3, 4)]
```

Edgelist with data in Python dictionary representation:

```
>>> lines = ["1 2 {'weight':3}", "2 3 {'weight':27}", "3 4 {'weight':3.0}"]
>>> G = bipartite.parse_edgelist(lines, nodetype=int)
>>> sorted(G.nodes())
[1, 2, 3, 4]
>>> sorted(G.edges(data=True))
[(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})]
```

read_edgelist

read_edgelist(*path*, *comments='#'*, *delimiter=None*, *create_using=None*, *nodetype=None*, *data=True*, *edgetype=None*, *encoding='utf-8'*)

Read a bipartite graph from a list of edges.

Parameters

path
: [file or string] File or filename to read. If a file is provided, it must be opened in 'rb' mode. Filenames ending in .gz or .bz2 will be uncompressed.

comments
: [string, optional] The character used to indicate the start of a comment.

delimiter
: [string, optional] The string used to separate values. The default is whitespace.

create_using
: [Graph container, optional,] Use specified container to build graph. The default is networkx.Graph, an undirected graph.

nodetype
: [int, float, str, Python type, optional] Convert node data from strings to specified type

data
: [bool or list of (label,type) tuples] Tuples specifying dictionary key names and types for edge data

edgetype
: [int, float, str, Python type, optional OBSOLETE] Convert edge data from strings to specified type and use as 'weight'

encoding: string, optional
: Specify which encoding to use when reading file.

Returns

G
: [graph] A networkx Graph or other type specified with create_using

See also:

parse_edgelist

Notes

Since nodes must be hashable, the function nodetype must return hashable types (e.g. int, float, str, frozenset - or tuples of those, etc.)

Examples

```
>>> from networkx.algorithms import bipartite
>>> G = nx.path_graph(4)
>>> G.add_nodes_from([0, 2], bipartite=0)
>>> G.add_nodes_from([1, 3], bipartite=1)
>>> bipartite.write_edgelist(G, "test.edgelist")
>>> G = bipartite.read_edgelist("test.edgelist")
```

```
>>> fh = open("test.edgelist", "rb")
>>> G = bipartite.read_edgelist(fh)
>>> fh.close()
```

```
>>> G = bipartite.read_edgelist("test.edgelist", nodetype=int)
```

Edgelist with data in a list:

```
>>> textline = "1 2 3"
>>> fh = open("test.edgelist", "w")
>>> d = fh.write(textline)
>>> fh.close()
>>> G = bipartite.read_edgelist(
...     "test.edgelist", nodetype=int, data=(("weight", float),)
... )
>>> list(G)
[1, 2]
>>> list(G.edges(data=True))
[(1, 2, {'weight': 3.0})]
```

See parse_edgelist() for more examples of formatting.

3.4.3 Matching

Provides functions for computing maximum cardinality matchings and minimum weight full matchings in a bipartite graph.

If you don't care about the particular implementation of the maximum matching algorithm, simply use the *maximum_matching()*. If you do care, you can import one of the named maximum matching algorithms directly.

For example, to find a maximum matching in the complete bipartite graph with two vertices on the left and three vertices on the right:

```
>>> G = nx.complete_bipartite_graph(2, 3)
>>> left, right = nx.bipartite.sets(G)
>>> list(left)
[0, 1]
>>> list(right)
[2, 3, 4]
>>> nx.bipartite.maximum_matching(G)
{0: 2, 1: 3, 2: 0, 3: 1}
```

The dictionary returned by *maximum_matching()* includes a mapping for vertices in both the left and right vertex sets.

Similarly, *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.

eppstein_matching(G[, top_nodes])	Returns the maximum cardinality matching of the bipartite graph `G`.
hopcroft_karp_matching(G[, top_nodes])	Returns the maximum cardinality matching of the bipartite graph `G`.
to_vertex_cover(G, matching[, top_nodes])	Returns the minimum vertex cover corresponding to the given maximum matching of the bipartite graph `G`.
maximum_matching(G[, top_nodes])	Returns the maximum cardinality matching in the given bipartite graph.
minimum_weight_full_matching(G[, top_nodes, ...])	Returns a minimum weight full matching of the bipartite graph `G`.

eppstein_matching

eppstein_matching(*G*, *top_nodes=None*)

Returns the maximum cardinality matching of the bipartite graph `G`.

Parameters

G
[NetworkX graph] Undirected bipartite graph

top_nodes
[container] 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, `matching`, such that `matching[v] == w` if node `v` is matched to node `w`. Unmatched nodes do not occur as a key in `matching`.

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.

See also:

hopcroft_karp_matching

Notes

This function is implemented with David Eppstein's version of the algorithm Hopcroft–Karp algorithm (see *hopcroft_karp_matching()*), which originally appeared in the Python Algorithms and Data Structures library (PADS).

See *bipartite documentation* for further details on how bipartite graphs are handled in NetworkX.

hopcroft_karp_matching

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.

See also:

maximum_matching
hopcroft_karp_matching
eppstein_matching

Notes

This function is implemented with the Hopcroft–Karp matching algorithm for bipartite graphs.

See *bipartite documentation* for further details on how bipartite graphs are handled in NetworkX.

References

[1]

to_vertex_cover

to_vertex_cover(*G*, *matching*, *top_nodes=None*)

Returns the minimum vertex cover corresponding to the given maximum matching of the bipartite graph `G`.

Parameters

G
[NetworkX graph] Undirected bipartite graph

matching
[dictionary] A dictionary whose keys are vertices in `G` and whose values are the distinct neighbors comprising the maximum matching for `G`, as returned by, for example, *maximum_matching()*. The dictionary *must* represent the maximum matching.

top_nodes
[container] 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

vertex_cover
[`set`] The minimum vertex cover in `G`.

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 using the procedure guaranteed by Konig's theorem, which proves an equivalence between a maximum matching and a minimum vertex cover in bipartite graphs.

Since a minimum vertex cover is the complement of a maximum independent set for any graph, one can compute the maximum independent set of a bipartite graph this way:

```
>>> G = nx.complete_bipartite_graph(2, 3)
>>> matching = nx.bipartite.maximum_matching(G)
>>> vertex_cover = nx.bipartite.to_vertex_cover(G, matching)
>>> independent_set = set(G) - vertex_cover
>>> print(list(independent_set))
[2, 3, 4]
```

See *bipartite documentation* for further details on how bipartite graphs are handled in NetworkX.

maximum_matching

maximum_matching(*G*, *top_nodes=None*)

Returns the maximum cardinality matching in the given bipartite graph.

This function is simply an alias for *hopcroft_karp_matching()*.

minimum_weight_full_matching

minimum_weight_full_matching(*G*, *top_nodes=None*, *weight='weight'*)

Returns a minimum weight full matching of the bipartite graph `G`.

Let $G = ((U, V), E)$ be a weighted bipartite graph with real weights $w : E \to \mathbb{R}$. This function then produces a matching $M \subseteq E$ with cardinality

$$|M| = \min(|U|, |V|),$$

which minimizes the sum of the weights of the edges included in the matching, $\sum_{e \in M} w(e)$, or raises an error if no such matching exists.

When $|U| = |V|$, this is commonly referred to as a perfect matching; here, since we allow $|U|$ and $|V|$ to differ, we follow Karp [1] and refer to the matching as *full*.

Parameters

G
[NetworkX graph] Undirected bipartite graph

top_nodes
[container] Container with all nodes in one bipartite node set. If not supplied it will be computed.

weight
[string, optional (default='weight')] The edge data key used to provide each value in the matrix.

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

ValueError
Raised if no full matching exists.

ImportError
Raised if SciPy is not available.

Notes

The problem of determining a minimum weight full matching is also known as the rectangular linear assignment problem. This implementation defers the calculation of the assignment to SciPy.

References

[1]

3.4.4 Matrix

biadjacency_matrix(G, row_order[, ...])	Returns the biadjacency matrix of the bipartite graph G.
from_biadjacency_matrix(A[, create_using, ...])	Creates a new bipartite graph from a biadjacency matrix given as a SciPy sparse array.

biadjacency_matrix

biadjacency_matrix(*G*, *row_order*, *column_order=None*, *dtype=None*, *weight='weight'*, *format='csr'*)

Returns the biadjacency matrix of the bipartite graph G.

Let `G = (U, V, E)` be a bipartite graph with node sets `U = u_{1},...,u_{r}` and `V = v_{1},...,v_{s}`. The biadjacency matrix [1] is the `r` x `s` matrix `B` in which `b_{i,j} = 1` if, and only if, `(u_i, v_j) in E`. If the parameter `weight` is not `None` and matches the name of an edge attribute, its value is used instead of 1.

Parameters

G
[graph] A NetworkX graph

row_order
[list of nodes] The rows of the matrix are ordered according to the list of nodes.

column_order
[list, optional] The columns of the matrix are ordered according to the list of nodes. If column_order is None, then the ordering of columns is arbitrary.

dtype
[NumPy data-type, optional] A valid NumPy dtype used to initialize the array. If None, then the NumPy default is used.

weight
[string or None, optional (default='weight')] The edge data key used to provide each value in the matrix. If None, then each edge has weight 1.

format
[str in {'bsr', 'csr', 'csc', 'coo', 'lil', 'dia', 'dok'}] The type of the matrix to be returned (default 'csr'). For some algorithms different implementations of sparse matrices can perform better. See [2] for details.

Returns

M
[SciPy sparse array] Biadjacency matrix representation of the bipartite graph G.

See also:

adjacency_matrix
from_biadjacency_matrix

Notes

No attempt is made to check that the input graph is bipartite.

For directed bipartite graphs only successors are considered as neighbors. To obtain an adjacency matrix with ones (or weight values) for both predecessors and successors you have to generate two biadjacency matrices where the rows of one of them are the columns of the other, and then add one to the transpose of the other.

References

[1], [2]

from_biadjacency_matrix

from_biadjacency_matrix(*A*, *create_using=None*, *edge_attribute='weight'*)

Creates a new bipartite graph from a biadjacency matrix given as a SciPy sparse array.

Parameters

A: scipy sparse array
A biadjacency matrix representation of a graph

create_using: NetworkX graph
Use specified graph for result. The default is Graph()

edge_attribute: string
Name of edge attribute to store matrix numeric value. The data will have the same type as the matrix entry (int, float, (real,imag)).

See also:

biadjacency_matrix
from_numpy_array

Notes

The nodes are labeled with the attribute `bipartite` set to an integer 0 or 1 representing membership in part 0 or part 1 of the bipartite graph.

If `create_using` is an instance of *networkx.MultiGraph* or *networkx.MultiDiGraph* and the entries of `A` are of type `int`, then this function returns a multigraph (of the same type as `create_using`) with parallel edges. In this case, `edge_attribute` will be ignored.

References

[1] https://en.wikipedia.org/wiki/Adjacency_matrix#Adjacency_matrix_of_a_bipartite_graph

3.4.5 Projections

One-mode (unipartite) projections of bipartite graphs.

projected_graph(B, nodes[, multigraph])	Returns the projection of B onto one of its node sets.
weighted_projected_graph(B, nodes[, ratio])	Returns a weighted projection of B onto one of its node sets.
collaboration_weighted_projected_graph(B, nodes)	Newman's weighted projection of B onto one of its node sets.
overlap_weighted_projected_graph(B, nodes[, ...])	Overlap weighted projection of B onto one of its node sets.
generic_weighted_projected_graph(B, nodes[, ...])	Weighted projection of B with a user-specified weight function.

projected_graph

projected_graph(*B*, *nodes*, *multigraph=False*)

Returns the projection of B onto one of its node sets.

Returns the graph G that is the projection of the bipartite graph B onto the specified nodes. They retain their attributes and are connected in G if they have a common neighbor in B.

Parameters

B
[NetworkX graph] The input graph should be bipartite.

nodes
[list or iterable] Nodes to project onto (the "bottom" nodes).

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.

See also:

is_bipartite
is_bipartite_node_set
sets
weighted_projected_graph
collaboration_weighted_projected_graph
overlap_weighted_projected_graph
generic_weighted_projected_graph

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 *bipartite documentation* for further details on how bipartite graphs are handled in NetworkX.

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']]
```

weighted_projected_graph

weighted_projected_graph(*B*, *nodes*, *ratio=False*)

Returns a weighted projection of B onto one of its node sets.

The weighted projected graph is the projection of the bipartite network B onto the specified nodes with weights representing the number of shared neighbors or the ratio between actual shared neighbors and possible shared neighbors if `ratio is True` [1]. The nodes retain their attributes and are connected in the resulting graph if they have an edge to a common node in the original graph.

Parameters

B
[NetworkX graph] The input graph should be bipartite.

nodes
[list or iterable] Distinct nodes to project onto (the "bottom" nodes).

ratio: Bool (default=False)
If True, edge weight is the ratio between actual shared neighbors and maximum possible shared neighbors (i.e., the size of the other node set). If False, edges weight is the number of shared neighbors.

Returns

Graph
[NetworkX graph] A graph that is the projection onto the given nodes.

See also:

is_bipartite
is_bipartite_node_set
sets
collaboration_weighted_projected_graph
overlap_weighted_projected_graph
generic_weighted_projected_graph
projected_graph

Notes

No attempt is made to verify that the input graph B is bipartite, or that the input nodes are distinct. However, if the length of the input nodes is greater than or equal to the nodes in the graph B, an exception is raised. If the nodes are not distinct but don't raise this error, the output weights will be incorrect. The graph and node properties are (shallow) copied to the projected graph.

See *bipartite documentation* for further details on how bipartite graphs are handled in NetworkX.

References

[1]

Examples

```
>>> from networkx.algorithms import bipartite
>>> B = nx.path_graph(4)
>>> G = bipartite.weighted_projected_graph(B, [1, 3])
>>> list(G)
[1, 3]
>>> list(G.edges(data=True))
[(1, 3, {'weight': 1})]
>>> G = bipartite.weighted_projected_graph(B, [1, 3], ratio=True)
>>> list(G.edges(data=True))
[(1, 3, {'weight': 0.5})]
```

collaboration_weighted_projected_graph

collaboration_weighted_projected_graph(*B*, *nodes*)

Newman's weighted projection of B onto one of its node sets.

The collaboration weighted projection is the projection of the bipartite network B onto the specified nodes with weights assigned using Newman's collaboration model [1]:

$$w_{u,v} = \sum_k \frac{\delta_u^k \delta_v^k}{d_k - 1}$$

where `u` and `v` are nodes from the bottom bipartite node set, and `k` is a node of the top node set. The value `d_k` is the degree of node `k` in the bipartite network and `delta_{u}^{k}` is 1 if node `u` is linked to node `k` in the original bipartite graph or 0 otherwise.

The nodes retain their attributes and are connected in the resulting graph if have an edge to a common node in the original bipartite graph.

Parameters

B
[NetworkX graph] The input graph should be bipartite.

nodes
[list or iterable] Nodes to project onto (the "bottom" nodes).

Returns

Graph
[NetworkX graph] A graph that is the projection onto the given nodes.

See also:

is_bipartite
is_bipartite_node_set
sets
weighted_projected_graph
overlap_weighted_projected_graph
generic_weighted_projected_graph
projected_graph

Notes

No attempt is made to verify that the input graph B is bipartite. The graph and node properties are (shallow) copied to the projected graph.

See *bipartite documentation* for further details on how bipartite graphs are handled in NetworkX.

References

[1]

Examples

```
>>> from networkx.algorithms import bipartite
>>> B = nx.path_graph(5)
>>> B.add_edge(1, 5)
>>> G = bipartite.collaboration_weighted_projected_graph(B, [0, 2, 4, 5])
>>> list(G)
[0, 2, 4, 5]
>>> for edge in sorted(G.edges(data=True)):
...     print(edge)
...
(0, 2, {'weight': 0.5})
(0, 5, {'weight': 0.5})
(2, 4, {'weight': 1.0})
(2, 5, {'weight': 0.5})
```

overlap_weighted_projected_graph

overlap_weighted_projected_graph(*B*, *nodes*, *jaccard=True*)

Overlap weighted projection of B onto one of its node sets.

The overlap weighted projection is the projection of the bipartite network B onto the specified nodes with weights representing the Jaccard index between the neighborhoods of the two nodes in the original bipartite network [1]:

$$w_{v,u} = \frac{|N(u) \cap N(v)|}{|N(u) \cup N(v)|}$$

or if the parameter 'jaccard' is False, the fraction of common neighbors by minimum of both nodes degree in the original bipartite graph [1]:

$$w_{v,u} = \frac{|N(u) \cap N(v)|}{min(|N(u)|, |N(v)|)}$$

The nodes retain their attributes and are connected in the resulting graph if have an edge to a common node in the original bipartite graph.

Parameters

B
[NetworkX graph] The input graph should be bipartite.

nodes
[list or iterable] Nodes to project onto (the "bottom" nodes).

jaccard: Bool (default=True)

Returns

Graph
[NetworkX graph] A graph that is the projection onto the given nodes.

See also:

is_bipartite
is_bipartite_node_set
sets
weighted_projected_graph
collaboration_weighted_projected_graph
generic_weighted_projected_graph
projected_graph

Notes

No attempt is made to verify that the input graph B is bipartite. The graph and node properties are (shallow) copied to the projected graph.

See *bipartite documentation* for further details on how bipartite graphs are handled in NetworkX.

References

[1]

Examples

```
>>> from networkx.algorithms import bipartite
>>> B = nx.path_graph(5)
>>> nodes = [0, 2, 4]
>>> G = bipartite.overlap_weighted_projected_graph(B, nodes)
>>> list(G)
[0, 2, 4]
>>> list(G.edges(data=True))
[(0, 2, {'weight': 0.5}), (2, 4, {'weight': 0.5})]
>>> G = bipartite.overlap_weighted_projected_graph(B, nodes, jaccard=False)
>>> list(G.edges(data=True))
[(0, 2, {'weight': 1.0}), (2, 4, {'weight': 1.0})]
```

generic_weighted_projected_graph

generic_weighted_projected_graph(*B*, *nodes*, *weight_function=None*)

Weighted projection of B with a user-specified weight function.

The bipartite network B is projected on to the specified nodes with weights computed by a user-specified function. This function must accept as a parameter the neighborhood sets of two nodes and return an integer or a float.

The nodes retain their attributes and are connected in the resulting graph if they have an edge to a common node in the original graph.

Parameters

B
: [NetworkX graph] The input graph should be bipartite.

nodes
: [list or iterable] Nodes to project onto (the "bottom" nodes).

weight_function
: [function] This function must accept as parameters the same input graph that this function, and two nodes; and return an integer or a float. The default function computes the number of shared neighbors.

Returns

Graph
: [NetworkX graph] A graph that is the projection onto the given nodes.

See also:

is_bipartite
is_bipartite_node_set
sets
weighted_projected_graph
collaboration_weighted_projected_graph
overlap_weighted_projected_graph
projected_graph

Notes

No attempt is made to verify that the input graph B is bipartite. The graph and node properties are (shallow) copied to the projected graph.

See *bipartite documentation* for further details on how bipartite graphs are handled in NetworkX.

Examples

```
>>> from networkx.algorithms import bipartite
>>> # Define some custom weight functions
>>> def jaccard(G, u, v):
...     unbrs = set(G[u])
...     vnbrs = set(G[v])
...     return float(len(unbrs & vnbrs)) / len(unbrs | vnbrs)
...
>>> def my_weight(G, u, v, weight="weight"):
...     w = 0
...     for nbr in set(G[u]) & set(G[v]):
...         w += G[u][nbr].get(weight, 1) + G[v][nbr].get(weight, 1)
...     return w
...
>>> # A complete bipartite graph with 4 nodes and 4 edges
>>> B = nx.complete_bipartite_graph(2, 2)
>>> # Add some arbitrary weight to the edges
>>> for i, (u, v) in enumerate(B.edges()):
...     B.edges[u, v]["weight"] = i + 1
...
>>> for edge in B.edges(data=True):
...     print(edge)
...
(0, 2, {'weight': 1})
(0, 3, {'weight': 2})
(1, 2, {'weight': 3})
(1, 3, {'weight': 4})
>>> # By default, the weight is the number of shared neighbors
>>> G = bipartite.generic_weighted_projected_graph(B, [0, 1])
>>> print(list(G.edges(data=True)))
[(0, 1, {'weight': 2})]
>>> # To specify a custom weight function use the weight_function parameter
>>> G = bipartite.generic_weighted_projected_graph(
...     B, [0, 1], weight_function=jaccard
... )
>>> print(list(G.edges(data=True)))
[(0, 1, {'weight': 1.0})]
>>> G = bipartite.generic_weighted_projected_graph(
...     B, [0, 1], weight_function=my_weight
... )
>>> print(list(G.edges(data=True)))
[(0, 1, {'weight': 10})]
```

3.4.6 Spectral

Spectral bipartivity measure.

spectral_bipartivity(G[, nodes, weight])	Returns the spectral bipartivity.

spectral_bipartivity

spectral_bipartivity(*G*, *nodes=None*, *weight='weight'*)

Returns the spectral bipartivity.

Parameters

G
[NetworkX graph]

nodes
[list or container optional(default is all nodes)] Nodes to return value of spectral bipartivity contribution.

weight
[string or None optional (default = 'weight')] Edge data key to use for edge weights. If None, weights set to 1.

Returns

sb
[float or dict] A single number if the keyword nodes is not specified, or a dictionary keyed by node with the spectral bipartivity contribution of that node as the value.

See also:

color

Notes

This implementation uses Numpy (dense) matrices which are not efficient for storing large sparse graphs.

References

[1]

Examples

```
>>> from networkx.algorithms import bipartite
>>> G = nx.path_graph(4)
>>> bipartite.spectral_bipartivity(G)
1.0
```

3.4.7 Clustering

Functions for computing clustering of pairs

clustering(G[, nodes, mode])	Compute a bipartite clustering coefficient for nodes.
average_clustering(G[, nodes, mode])	Compute the average bipartite clustering coefficient.
latapy_clustering(G[, nodes, mode])	Compute a bipartite clustering coefficient for nodes.
robins_alexander_clustering(G)	Compute the bipartite clustering of G.

clustering

clustering(*G*, *nodes=None*, *mode='dot'*)

Compute a bipartite clustering coefficient for nodes.

The bipartie clustering coefficient is a measure of local density of connections defined as [1]:

$$c_u = \frac{\sum_{v \in N(N(u))} c_{uv}}{|N(N(u))|}$$

where `N(N(u))` are the second order neighbors of `u` in `G` excluding `u`, and `c_{uv}` is the pairwise clustering coefficient between nodes `u` and `v`.

The mode selects the function for `c_{uv}` which can be:

`dot`:

$$c_{uv} = \frac{|N(u) \cap N(v)|}{|N(u) \cup N(v)|}$$

`min`:

$$c_{uv} = \frac{|N(u) \cap N(v)|}{min(|N(u)|, |N(v)|)}$$

`max`:

$$c_{uv} = \frac{|N(u) \cap N(v)|}{max(|N(u)|, |N(v)|)}$$

Parameters

G
[graph] A bipartite graph

nodes
[list or iterable (optional)] Compute bipartite clustering for these nodes. The default is all nodes in G.

mode
[string] The pariwise bipartite clustering method to be used in the computation. It must be "dot", "max", or "min".

Returns

clustering
[dictionary] A dictionary keyed by node with the clustering coefficient value.

See also:

robins_alexander_clustering
average_clustering
networkx.algorithms.cluster.square_clustering

References

[1]

Examples

```
>>> from networkx.algorithms import bipartite
>>> G = nx.path_graph(4)  # path graphs are bipartite
>>> c = bipartite.clustering(G)
>>> c[0]
0.5
>>> c = bipartite.clustering(G, mode="min")
>>> c[0]
1.0
```

average_clustering

average_clustering(*G*, *nodes=None*, *mode='dot'*)

Compute the average bipartite clustering coefficient.

A clustering coefficient for the whole graph is the average,

$$C = \frac{1}{n}\sum_{v \in G} c_v,$$

where `n` is the number of nodes in `G`.

Similar measures for the two bipartite sets can be defined [1]

$$C_X = \frac{1}{|X|}\sum_{v \in X} c_v,$$

where `X` is a bipartite set of `G`.

Parameters

G
: [graph] a bipartite graph

nodes
: [list or iterable, optional] A container of nodes to use in computing the average. The nodes should be either the entire graph (the default) or one of the bipartite sets.

mode
: [string] The pariwise bipartite clustering method. It must be “dot”, “max”, or “min”

Returns

clustering
: [float] The average bipartite clustering for the given set of nodes or the entire graph if no nodes are specified.

See also:

clustering

Notes

The container of nodes passed to this function must contain all of the nodes in one of the bipartite sets ("top" or "bottom") in order to compute the correct average bipartite clustering coefficients. See *bipartite documentation* for further details on how bipartite graphs are handled in NetworkX.

References

[1]

Examples

```
>>> from networkx.algorithms import bipartite
>>> G = nx.star_graph(3)  # star graphs are bipartite
>>> bipartite.average_clustering(G)
0.75
>>> X, Y = bipartite.sets(G)
>>> bipartite.average_clustering(G, X)
0.0
>>> bipartite.average_clustering(G, Y)
1.0
```

latapy_clustering

latapy_clustering(*G*, *nodes=None*, *mode='dot'*)

Compute a bipartite clustering coefficient for nodes.

The bipartie clustering coefficient is a measure of local density of connections defined as [1]:

$$c_u = \frac{\sum_{v \in N(N(u))} c_{uv}}{|N(N(u))|}$$

where `N(N(u))` are the second order neighbors of `u` in `G` excluding `u`, and `c_{uv}` is the pairwise clustering coefficient between nodes `u` and `v`.

The mode selects the function for `c_{uv}` which can be:

`dot`:

$$c_{uv} = \frac{|N(u) \cap N(v)|}{|N(u) \cup N(v)|}$$

`min`:

$$c_{uv} = \frac{|N(u) \cap N(v)|}{min(|N(u)|, |N(v)|)}$$

`max`:

$$c_{uv} = \frac{|N(u) \cap N(v)|}{max(|N(u)|, |N(v)|)}$$

Parameters

G
[graph] A bipartite graph

nodes
[list or iterable (optional)] Compute bipartite clustering for these nodes. The default is all nodes in G.

mode
[string] The pariwise bipartite clustering method to be used in the computation. It must be "dot", "max", or "min".

Returns

clustering
[dictionary] A dictionary keyed by node with the clustering coefficient value.

See also:

robins_alexander_clustering
average_clustering
networkx.algorithms.cluster.square_clustering

References

[1]

Examples

```
>>> from networkx.algorithms import bipartite
>>> G = nx.path_graph(4)  # path graphs are bipartite
>>> c = bipartite.clustering(G)
>>> c[0]
0.5
>>> c = bipartite.clustering(G, mode="min")
>>> c[0]
1.0
```

robins_alexander_clustering

robins_alexander_clustering(*G*)

Compute the bipartite clustering of G.

Robins and Alexander [1] defined bipartite clustering coefficient as four times the number of four cycles `C_4` divided by the number of three paths `L_3` in a bipartite graph:

$$CC_4 = \frac{4 * C_4}{L_3}$$

Parameters

G
[graph] a bipartite graph

Returns

clustering
[float] The Robins and Alexander bipartite clustering for the input graph.

See also:

latapy_clustering
networkx.algorithms.cluster.square_clustering

References

[1]

Examples

```
>>> from networkx.algorithms import bipartite
>>> G = nx.davis_southern_women_graph()
>>> print(round(bipartite.robins_alexander_clustering(G), 3))
0.468
```

3.4.8 Redundancy

Node redundancy for bipartite graphs.

node_redundancy(G[, nodes])	Computes the node redundancy coefficients for the nodes in the bipartite graph G.

node_redundancy

node_redundancy(*G*, *nodes=None*)

Computes the node redundancy coefficients for the nodes in the bipartite graph G.

The redundancy coefficient of a node v is the fraction of pairs of neighbors of v that are both linked to other nodes. In a one-mode projection these nodes would be linked together even if v were not there.

More formally, for any vertex v, the *redundancy coefficient of `v`* is defined by

$$rc(v) = \frac{|\{\{u, w\} \subseteq N(v),\ \exists v' \neq v,\ (v', u) \in E \text{ and } (v', w) \in E\}|}{\frac{|N(v)|(|N(v)|-1)}{2}},$$

where N(v) is the set of neighbors of v in G.

Parameters

G
[graph] A bipartite graph

nodes
[list or iterable (optional)] Compute redundancy for these nodes. The default is all nodes in G.

Returns

redundancy
[dictionary] A dictionary keyed by node with the node redundancy value.

Raises

NetworkXError
If any of the nodes in the graph (or in nodes, if specified) has (out-)degree less than two (which would result in division by zero, according to the definition of the redundancy coefficient).

References

[1]

Examples

Compute the redundancy coefficient of each node in a graph:

```
>>> from networkx.algorithms import bipartite
>>> G = nx.cycle_graph(4)
>>> rc = bipartite.node_redundancy(G)
>>> rc[0]
1.0
```

Compute the average redundancy for the graph:

```
>>> from networkx.algorithms import bipartite
>>> G = nx.cycle_graph(4)
>>> rc = bipartite.node_redundancy(G)
>>> sum(rc.values()) / len(G)
1.0
```

Compute the average redundancy for a set of nodes:

```
>>> from networkx.algorithms import bipartite
>>> G = nx.cycle_graph(4)
>>> rc = bipartite.node_redundancy(G)
>>> nodes = [0, 2]
>>> sum(rc[n] for n in nodes) / len(nodes)
1.0
```

3.4.9 Centrality

closeness_centrality(G, nodes[, normalized])	Compute the closeness centrality for nodes in a bipartite network.
degree_centrality(G, nodes)	Compute the degree centrality for nodes in a bipartite network.
betweenness_centrality(G, nodes)	Compute betweenness centrality for nodes in a bipartite network.

closeness_centrality

closeness_centrality(*G*, *nodes*, *normalized=True*)

Compute the closeness centrality for nodes in a bipartite network.

The closeness of a node is the distance to all other nodes in the graph or in the case that the graph is not connected to all other nodes in the connected component containing that node.

Parameters

G

[graph] A bipartite network

nodes
[list or container] Container with all nodes in one bipartite node set.

normalized
[bool, optional] If True (default) normalize by connected component size.

Returns

closeness
[dictionary] Dictionary keyed by node with bipartite closeness centrality as the value.

See also:

betweenness_centrality
degree_centrality
sets()
is_bipartite()

Notes

The nodes input parameter must contain all nodes in one bipartite node set, but the dictionary returned contains all nodes from both node sets. See *bipartite documentation* for further details on how bipartite graphs are handled in NetworkX.

Closeness centrality is normalized by the minimum distance possible. In the bipartite case the minimum distance for a node in one bipartite node set is 1 from all nodes in the other node set and 2 from all other nodes in its own set [1]. Thus the closeness centrality for node `v` in the two bipartite sets `U` with `n` nodes and `V` with `m` nodes is

$$c_v = \frac{m + 2(n-1)}{d}, \text{for} v \in U,$$
$$c_v = \frac{n + 2(m-1)}{d}, \text{for} v \in V,$$

where `d` is the sum of the distances from `v` to all other nodes.

Higher values of closeness indicate higher centrality.

As in the unipartite case, setting normalized=True causes the values to normalized further to n-1 / size(G)-1 where n is the number of nodes in the connected part of graph containing the node. If the graph is not completely connected, this algorithm computes the closeness centrality for each connected part separately.

References

[1]

degree_centrality

degree_centrality(*G*, *nodes*)

Compute the degree centrality for nodes in a bipartite network.

The degree centrality for a node `v` is the fraction of nodes connected to it.

Parameters

G
[graph] A bipartite network

nodes
[list or container] Container with all nodes in one bipartite node set.

Returns

centrality
[dictionary] Dictionary keyed by node with bipartite degree centrality as the value.

See also:

betweenness_centrality
closeness_centrality
sets()
is_bipartite()

Notes

The nodes input parameter must contain all nodes in one bipartite node set, but the dictionary returned contains all nodes from both bipartite node sets. See *bipartite documentation* for further details on how bipartite graphs are handled in NetworkX.

For unipartite networks, the degree centrality values are normalized by dividing by the maximum possible degree (which is `n-1` where `n` is the number of nodes in G).

In the bipartite case, the maximum possible degree of a node in a bipartite node set is the number of nodes in the opposite node set [1]. The degree centrality for a node `v` in the bipartite sets `U` with `n` nodes and `V` with `m` nodes is

$$d_v = \frac{deg(v)}{m}, \text{for} v \in U,$$
$$d_v = \frac{deg(v)}{n}, \text{for} v \in V,$$

where `deg(v)` is the degree of node `v`.

References

[1]

betweenness_centrality

betweenness_centrality(*G*, *nodes*)

Compute betweenness centrality for nodes in a bipartite network.

Betweenness centrality of a node `v` is the sum of the fraction of all-pairs shortest paths that pass through `v`.

Values of betweenness are normalized by the maximum possible value which for bipartite graphs is limited by the relative size of the two node sets [1].

Let `n` be the number of nodes in the node set `U` and `m` be the number of nodes in the node set `V`, then nodes in `U` are normalized by dividing by

$$\frac{1}{2}[m^2(s+1)^2 + m(s+1)(2t-s-1) - t(2s-t+3)],$$

where

$$s = (n-1) \div m, t = (n-1) \mod m,$$

and nodes in V are normalized by dividing by

$$\frac{1}{2}[n^2(p+1)^2 + n(p+1)(2r-p-1) - r(2p-r+3)],$$

where,

$$p = (m-1) \div n, r = (m-1) \mod n.$$

Parameters

G
[graph] A bipartite graph

nodes
[list or container] Container with all nodes in one bipartite node set.

Returns

betweenness
[dictionary] Dictionary keyed by node with bipartite betweenness centrality as the value.

See also:

degree_centrality
closeness_centrality
sets()
is_bipartite()

Notes

The nodes input parameter must contain all nodes in one bipartite node set, but the dictionary returned contains all nodes from both node sets. See *bipartite documentation* for further details on how bipartite graphs are handled in NetworkX.

References

[1]

3.4.10 Generators

Generators and functions for bipartite graphs.

complete_bipartite_graph(n1, n2[, create_using])	Returns the complete bipartite graph K_{n_1,n_2}.
configuration_model(aseq, bseq[, ...])	Returns a random bipartite graph from two given degree sequences.
havel_hakimi_graph(aseq, bseq[, create_using])	Returns a bipartite graph from two given degree sequences using a Havel-Hakimi style construction.
reverse_havel_hakimi_graph(aseq, bseq[, ...])	Returns a bipartite graph from two given degree sequences using a Havel-Hakimi style construction.
alternating_havel_hakimi_graph(aseq, bseq[, ...])	Returns a bipartite graph from two given degree sequences using an alternating Havel-Hakimi style construction.
preferential_attachment_graph(aseq, p[, ...])	Create a bipartite graph with a preferential attachment model from a given single degree sequence.
random_graph(n, m, p[, seed, directed])	Returns a bipartite random graph.
gnmk_random_graph(n, m, k[, seed, directed])	Returns a random bipartite graph G_{n,m,k}.

complete_bipartite_graph

complete_bipartite_graph(*n1*, *n2*, *create_using=None*)

Returns the complete bipartite graph K_{n_1,n_2}.

The graph is composed of two partitions with nodes 0 to (n1 - 1) in the first and nodes n1 to (n1 + n2 - 1) in the second. Each node in the first is connected to each node in the second.

Parameters

n1, n2
[integer or iterable container of nodes] If integers, nodes are from range(n1) and range(n1, n1 + n2). If a container, the elements are the nodes.

create_using
[NetworkX graph instance, (default: nx.Graph)] Return graph of this type.

Notes

Nodes are the integers 0 to n1 + n2 − 1 unless either n1 or n2 are containers of nodes. If only one of n1 or n2 are integers, that integer is replaced by range of that integer.

The nodes are assigned the attribute 'bipartite' with the value 0 or 1 to indicate which bipartite set the node belongs to.

This function is not imported in the main namespace. To use it use nx.bipartite.complete_bipartite_graph

configuration_model

configuration_model(*aseq*, *bseq*, *create_using=None*, *seed=None*)

Returns a random bipartite graph from two given degree sequences.

Parameters

aseq
[list] Degree sequence for node set A.

bseq
[list] Degree sequence for node set B.

create_using
[NetworkX graph instance, optional] Return graph of this type.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

The graph is composed of two partitions. Set A has nodes 0 to (len(aseq) - 1) and set B has nodes len(aseq) to (len(bseq) - 1). Nodes from set A are connected to nodes in set B by choosing randomly from the possible free stubs, one in A and one in B.

Notes

The sum of the two sequences must be equal: sum(aseq)=sum(bseq) If no graph type is specified use MultiGraph with parallel edges. If you want a graph with no parallel edges use create_using=Graph() but then the resulting degree sequences might not be exact.

The nodes are assigned the attribute 'bipartite' with the value 0 or 1 to indicate which bipartite set the node belongs to.

This function is not imported in the main namespace. To use it use nx.bipartite.configuration_model

havel_hakimi_graph

havel_hakimi_graph(*aseq*, *bseq*, *create_using=None*)

Returns a bipartite graph from two given degree sequences using a Havel-Hakimi style construction.

The graph is composed of two partitions. Set A has nodes 0 to (len(aseq) - 1) and set B has nodes len(aseq) to (len(bseq) - 1). Nodes from the set A are connected to nodes in the set B by connecting the highest degree nodes in set A to the highest degree nodes in set B until all stubs are connected.

Parameters

aseq
[list] Degree sequence for node set A.

bseq
[list] Degree sequence for node set B.

create_using
[NetworkX graph instance, optional] Return graph of this type.

Notes

The sum of the two sequences must be equal: sum(aseq)=sum(bseq) If no graph type is specified use MultiGraph with parallel edges. If you want a graph with no parallel edges use create_using=Graph() but then the resulting degree sequences might not be exact.

The nodes are assigned the attribute 'bipartite' with the value 0 or 1 to indicate which bipartite set the node belongs to.

This function is not imported in the main namespace. To use it use nx.bipartite.havel_hakimi_graph

reverse_havel_hakimi_graph

reverse_havel_hakimi_graph(*aseq*, *bseq*, *create_using=None*)

Returns a bipartite graph from two given degree sequences using a Havel-Hakimi style construction.

The graph is composed of two partitions. Set A has nodes 0 to (len(aseq) - 1) and set B has nodes len(aseq) to (len(bseq) - 1). Nodes from set A are connected to nodes in the set B by connecting the highest degree nodes in set A to the lowest degree nodes in set B until all stubs are connected.

Parameters

aseq
[list] Degree sequence for node set A.

bseq
[list] Degree sequence for node set B.

create_using
[NetworkX graph instance, optional] Return graph of this type.

Notes

The sum of the two sequences must be equal: sum(aseq)=sum(bseq) If no graph type is specified use MultiGraph with parallel edges. If you want a graph with no parallel edges use create_using=Graph() but then the resulting degree sequences might not be exact.

The nodes are assigned the attribute 'bipartite' with the value 0 or 1 to indicate which bipartite set the node belongs to.

This function is not imported in the main namespace. To use it use nx.bipartite.reverse_havel_hakimi_graph

alternating_havel_hakimi_graph

alternating_havel_hakimi_graph(*aseq*, *bseq*, *create_using=None*)

Returns a bipartite graph from two given degree sequences using an alternating Havel-Hakimi style construction.

The graph is composed of two partitions. Set A has nodes 0 to (len(aseq) - 1) and set B has nodes len(aseq) to (len(bseq) - 1). Nodes from the set A are connected to nodes in the set B by connecting the highest degree nodes in set A to alternatively the highest and the lowest degree nodes in set B until all stubs are connected.

Parameters

aseq
[list] Degree sequence for node set A.

bseq
[list] Degree sequence for node set B.

create_using
[NetworkX graph instance, optional] Return graph of this type.

Notes

The sum of the two sequences must be equal: sum(aseq)=sum(bseq) If no graph type is specified use MultiGraph with parallel edges. If you want a graph with no parallel edges use create_using=Graph() but then the resulting degree sequences might not be exact.

The nodes are assigned the attribute 'bipartite' with the value 0 or 1 to indicate which bipartite set the node belongs to.

This function is not imported in the main namespace. To use it use nx.bipartite.alternating_havel_hakimi_graph

preferential_attachment_graph

preferential_attachment_graph(*aseq*, *p*, *create_using=None*, *seed=None*)

Create a bipartite graph with a preferential attachment model from a given single degree sequence.

The graph is composed of two partitions. Set A has nodes 0 to (len(aseq) - 1) and set B has nodes starting with node len(aseq). The number of nodes in set B is random.

Parameters

aseq
[list] Degree sequence for node set A.

p
[float] Probability that a new bottom node is added.

create_using
[NetworkX graph instance, optional] Return graph of this type.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Notes

The nodes are assigned the attribute 'bipartite' with the value 0 or 1 to indicate which bipartite set the node belongs to.

This function is not imported in the main namespace. To use it use nx.bipartite.preferential_attachment_graph

References

[1], [2]

random_graph

random_graph(*n*, *m*, *p*, *seed=None*, *directed=False*)

Returns a bipartite random graph.

This is a bipartite version of the binomial (Erdős-Rényi) graph. The graph is composed of two partitions. Set A has nodes 0 to (n - 1) and set B has nodes n to (n + m - 1).

Parameters

n
[int] The number of nodes in the first bipartite set.

m
[int] The number of nodes in the second bipartite set.

p
[float] Probability for edge creation.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

directed
[bool, optional (default=False)] If True return a directed graph

See also:

gnp_random_graph, ***configuration_model***

Notes

The bipartite random graph algorithm chooses each of the n*m (undirected) or 2*nm (directed) possible edges with probability p.

This algorithm is $O(n + m)$ where m is the expected number of edges.

The nodes are assigned the attribute 'bipartite' with the value 0 or 1 to indicate which bipartite set the node belongs to.

This function is not imported in the main namespace. To use it use nx.bipartite.random_graph

References

[1]

gnmk_random_graph

gnmk_random_graph(*n*, *m*, *k*, *seed=None*, *directed=False*)

Returns a random bipartite graph G_{n,m,k}.

Produces a bipartite graph chosen randomly out of the set of all graphs with n top nodes, m bottom nodes, and k edges. The graph is composed of two sets of nodes. Set A has nodes 0 to (n - 1) and set B has nodes n to (n + m - 1).

Parameters

n
[int] The number of nodes in the first bipartite set.

m
[int] The number of nodes in the second bipartite set.

k
[int] The number of edges

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

directed
[bool, optional (default=False)] If True return a directed graph

See also:

gnm_random_graph

Notes

If k > m * n then a complete bipartite graph is returned.

This graph is a bipartite version of the `G_{nm}` random graph model.

The nodes are assigned the attribute 'bipartite' with the value 0 or 1 to indicate which bipartite set the node belongs to.

This function is not imported in the main namespace. To use it use nx.bipartite.gnmk_random_graph

Examples

from nx.algorithms import bipartite G = bipartite.gnmk_random_graph(10,20,50)

3.4.11 Covering

Functions related to graph covers.

min_edge_cover(G[, matching_algorithm])	Returns a set of edges which constitutes the minimum edge cover of the graph.

min_edge_cover

min_edge_cover(*G*, *matching_algorithm=None*)

Returns a set of edges which constitutes the minimum edge cover of the graph.

The smallest edge cover can be found in polynomial time by finding a maximum matching and extending it greedily so that all nodes are covered.

Parameters

G
[NetworkX graph] An undirected bipartite graph.

matching_algorithm
[function] A function that returns a maximum cardinality matching in a given bipartite graph. The function must take one input, the graph `G`, and return a dictionary mapping each node to its mate. If not specified, *hopcroft_karp_matching()* will be used. Other possibilities include *eppstein_matching()*,

Returns

set
A set of the edges in a minimum edge cover of the graph, given as pairs of nodes. It contains both the edges `(u, v)` and `(v, u)` for given nodes `u` and `v` among the edges of minimum edge cover.

Notes

An edge cover of a graph is a set of edges such that every node of the graph is incident to at least one edge of the set. A minimum edge cover is an edge covering of smallest cardinality.

Due to its implementation, the worst-case running time of this algorithm is bounded by the worst-case running time of the function `matching_algorithm`.

3.5 Boundary

Routines to find the boundary of a set of nodes.

An edge boundary is a set of edges, each of which has exactly one endpoint in a given set of nodes (or, in the case of directed graphs, the set of edges whose source node is in the set).

A node boundary of a set *S* of nodes is the set of (out-)neighbors of nodes in *S* that are outside *S*.

edge_boundary(G, nbunch1[, nbunch2, data, ...])	Returns the edge boundary of `nbunch1`.
node_boundary(G, nbunch1[, nbunch2])	Returns the node boundary of `nbunch1`.

3.5.1 edge_boundary

edge_boundary(*G*, *nbunch1*, *nbunch2=None*, *data=False*, *keys=False*, *default=None*)

Returns the edge boundary of `nbunch1`.

The *edge boundary* of a set *S* with respect to a set *T* is the set of edges (*u*, *v*) such that *u* is in *S* and *v* is in *T*. If *T* is not specified, it is assumed to be the set of all nodes not in *S*.

Parameters

G
[NetworkX graph]

nbunch1
[iterable] Iterable of nodes in the graph representing the set of nodes whose edge boundary will be returned. (This is the set *S* from the definition above.)

nbunch2
[iterable] Iterable of nodes representing the target (or "exterior") set of nodes. (This is the set *T* from the definition above.) If not specified, this is assumed to be the set of all nodes in `G` not in `nbunch1`.

keys
[bool] This parameter has the same meaning as in `MultiGraph.edges()`.

data
[bool or object] This parameter has the same meaning as in `MultiGraph.edges()`.

default
[object] This parameter has the same meaning as in `MultiGraph.edges()`.

Returns

iterator
An iterator over the edges in the boundary of `nbunch1` with respect to `nbunch2`. If `keys`, `data`, or `default` are specified and `G` is a multigraph, then edges are returned with keys and/or data, as in `MultiGraph.edges()`.

Notes

Any element of `nbunch` that is not in the graph `G` will be ignored.

`nbunch1` and `nbunch2` are usually meant to be disjoint, but in the interest of speed and generality, that is not required here.

3.5.2 node_boundary

node_boundary(*G*, *nbunch1*, *nbunch2=None*)

Returns the node boundary of `nbunch1`.

The *node boundary* of a set S with respect to a set T is the set of nodes v in T such that for some u in S, there is an edge joining u to v. If T is not specified, it is assumed to be the set of all nodes not in S.

Parameters

G
[NetworkX graph]

nbunch1
[iterable] Iterable of nodes in the graph representing the set of nodes whose node boundary will be returned. (This is the set S from the definition above.)

nbunch2
[iterable] Iterable of nodes representing the target (or "exterior") set of nodes. (This is the set T from the definition above.) If not specified, this is assumed to be the set of all nodes in `G` not in `nbunch1`.

Returns

set
The node boundary of `nbunch1` with respect to `nbunch2`.

Notes

Any element of `nbunch` that is not in the graph `G` will be ignored.

`nbunch1` and `nbunch2` are usually meant to be disjoint, but in the interest of speed and generality, that is not required here.

3.6 Bridges

Bridge-finding algorithms.

bridges(G[, root])	Generate all bridges in a graph.
has_bridges(G[, root])	Decide whether a graph has any bridges.
local_bridges(G[, with_span, weight])	Iterate over local bridges of `G` optionally computing the span

3.6.1 bridges

bridges(*G*, *root=None*)

Generate all bridges in a graph.

A *bridge* in a graph is an edge whose removal causes the number of connected components of the graph to increase. Equivalently, a bridge is an edge that does not belong to any cycle. Bridges are also known as cut-edges, isthmuses, or cut arcs.

Parameters

G
[undirected graph]

root
[node (optional)] A node in the graph `G`. If specified, only the bridges in the connected component containing this node will be returned.

Yields

e
[edge] An edge in the graph whose removal disconnects the graph (or causes the number of connected components to increase).

Raises

NodeNotFound
If `root` is not in the graph `G`.

NetworkXNotImplemented
If `G` is a directed graph.

Notes

This is an implementation of the algorithm described in [1]. An edge is a bridge if and only if it is not contained in any chain. Chains are found using the `networkx.chain_decomposition()` function.

The algorithm described in [1] requires a simple graph. If the provided graph is a multigraph, we convert it to a simple graph and verify that any bridges discovered by the chain decomposition algorithm are not multi-edges.

Ignoring polylogarithmic factors, the worst-case time complexity is the same as the `networkx.chain_decomposition()` function, $O(m + n)$, where n is the number of nodes in the graph and m is the number of edges.

References

[1]

Examples

The barbell graph with parameter zero has a single bridge:

```
>>> G = nx.barbell_graph(10, 0)
>>> list(nx.bridges(G))
[(9, 10)]
```

3.6.2 has_bridges

has_bridges(*G*, *root=None*)

Decide whether a graph has any bridges.

A *bridge* in a graph is an edge whose removal causes the number of connected components of the graph to increase.

Parameters

G
[undirected graph]

root
[node (optional)] A node in the graph `G`. If specified, only the bridges in the connected component containing this node will be considered.

Returns

bool
Whether the graph (or the connected component containing `root`) has any bridges.

Raises

NodeNotFound
If `root` is not in the graph `G`.

NetworkXNotImplemented
If `G` is a directed graph.

Notes

This implementation uses the `networkx.bridges()` function, so it shares its worst-case time complexity, $O(m+n)$, ignoring polylogarithmic factors, where n is the number of nodes in the graph and m is the number of edges.

Examples

The barbell graph with parameter zero has a single bridge:

```
>>> G = nx.barbell_graph(10, 0)
>>> nx.has_bridges(G)
True
```

On the other hand, the cycle graph has no bridges:

```
>>> G = nx.cycle_graph(5)
>>> nx.has_bridges(G)
False
```

3.6.3 local_bridges

local_bridges(*G*, *with_span=True*, *weight=None*)

Iterate over local bridges of `G` optionally computing the span

A *local bridge* is an edge whose endpoints have no common neighbors. That is, the edge is not part of a triangle in the graph.

The *span* of a *local bridge* is the shortest path length between the endpoints if the local bridge is removed.

Parameters

G
[undirected graph]

with_span
[bool] If True, yield a 3-tuple `(u, v, span)`

weight
[function, string or None (default: None)] If function, used to compute edge weights for the span. If string, the edge data attribute used in calculating span. If None, all edges have weight 1.

Yields

e
[edge] The local bridges as an edge 2-tuple of nodes `(u, v)` or as a 3-tuple `(u, v, span)` when `with_span is True`.

Raises

NetworkXNotImplemented
If `G` is a directed graph or multigraph.

Examples

A cycle graph has every edge a local bridge with span N-1.

```
>>> G = nx.cycle_graph(9)
>>> (0, 8, 8) in set(nx.local_bridges(G))
True
```

3.7 Centrality

3.7.1 Degree

degree_centrality(G)	Compute the degree centrality for nodes.
in_degree_centrality(G)	Compute the in-degree centrality for nodes.
out_degree_centrality(G)	Compute the out-degree centrality for nodes.

degree_centrality

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.

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.

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}
```

in_degree_centrality

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.

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.

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.

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}
```

out_degree_centrality

out_degree_centrality(*G*)

Compute the out-degree centrality for nodes.

The out-degree centrality for a node v is the fraction of nodes its outgoing edges are connected to.

Parameters

G
[graph] A NetworkX graph

Returns

nodes
[dictionary] Dictionary of nodes with out-degree centrality as values.

Raises

NetworkXNotImplemented
If G is undirected.

See also:

degree_centrality, *in_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.

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.

Examples

```
>>> G = nx.DiGraph([(0, 1), (0, 2), (0, 3), (1, 2), (1, 3)])
>>> nx.out_degree_centrality(G)
{0: 1.0, 1: 0.6666666666666666, 2: 0.0, 3: 0.0}
```

3.7.2 Eigenvector

eigenvector_centrality(G[, max_iter, tol, ...])	Compute the eigenvector centrality for the graph G.
eigenvector_centrality_numpy(G[, weight, ...])	Compute the eigenvector centrality for the graph G.
katz_centrality(G[, alpha, beta, max_iter, ...])	Compute the Katz centrality for the nodes of the graph G.
katz_centrality_numpy(G[, alpha, beta, ...])	Compute the Katz centrality for the graph G.

eigenvector_centrality

eigenvector_centrality(*G*, *max_iter=100*, *tol=1e-06*, *nstart=None*, *weight=None*)

Compute the eigenvector centrality for the graph `G`.

Eigenvector centrality computes the centrality for a node based on the centrality of its neighbors. The eigenvector centrality for node i is the i-th element of the vector x defined by the equation

$$Ax = \lambda x$$

where A is the adjacency matrix of the graph `G` with eigenvalue λ. By virtue of the Perron–Frobenius theorem, there is a unique solution x, all of whose entries are positive, if λ is the largest eigenvalue of the adjacency matrix A ([2]).

Parameters

G
[graph] A networkx graph

max_iter
[integer, optional (default=100)] Maximum number of iterations in power method.

tol
[float, optional (default=1.0e-6)] Error tolerance used to check convergence in power method iteration.

nstart
[dictionary, optional (default=None)] Starting value of eigenvector iteration for each node.

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. In this measure the weight is interpreted as the connection strength.

Returns

nodes
[dictionary] Dictionary of nodes with eigenvector centrality as the value.

Raises

NetworkXPointlessConcept
If the graph `G` is the null graph.

NetworkXError
If each value in `nstart` is zero.

PowerIterationFailedConvergence
If the algorithm fails to converge to the specified tolerance within the specified number of iterations of the power iteration method.

See also:

eigenvector_centrality_numpy
pagerank
hits

Notes

The measure was introduced by [1] and is discussed in [2].

The power iteration method is used to compute the eigenvector and convergence is **not** guaranteed. Our method stops after `max_iter` iterations or when the change in the computed vector between two iterations is smaller than an error tolerance of `G.number_of_nodes() * tol`. This implementation uses $(A + I)$ rather than the adjacency matrix A because it shifts the spectrum to enable discerning the correct eigenvector even for networks with multiple dominant eigenvalues.

For directed graphs this is "left" eigenvector centrality which corresponds to the in-edges in the graph. For out-edges eigenvector centrality first reverse the graph with `G.reverse()`.

References

[1], [2]

Examples

```
>>> G = nx.path_graph(4)
>>> centrality = nx.eigenvector_centrality(G)
>>> sorted((v, f"{c:0.2f}") for v, c in centrality.items())
[(0, '0.37'), (1, '0.60'), (2, '0.60'), (3, '0.37')]
```

eigenvector_centrality_numpy

eigenvector_centrality_numpy(*G*, *weight=None*, *max_iter=50*, *tol=0*)

Compute the eigenvector centrality for the graph G.

Eigenvector centrality computes the centrality for a node based on the centrality of its neighbors. The eigenvector centrality for node i is

$$Ax = \lambda x$$

where A is the adjacency matrix of the graph G with eigenvalue λ. By virtue of the Perron–Frobenius theorem, there is a unique and positive solution if λ is the largest eigenvalue associated with the eigenvector of the adjacency matrix A ([2]).

Parameters

G
: [graph] A networkx graph

weight
: [None or string, optional (default=None)] The name of the edge attribute used as weight. If None, all edge weights are considered equal. In this measure the weight is interpreted as the connection strength.

max_iter
: [integer, optional (default=100)] Maximum number of iterations in power method.

tol
: [float, optional (default=1.0e-6)] Relative accuracy for eigenvalues (stopping criterion). The default value of 0 implies machine precision.

Returns

nodes
[dictionary] Dictionary of nodes with eigenvector centrality as the value.

Raises

NetworkXPointlessConcept
If the graph `G` is the null graph.

See also:

eigenvector_centrality
pagerank
hits

Notes

The measure was introduced by [1].

This algorithm uses the SciPy sparse eigenvalue solver (ARPACK) to find the largest eigenvalue/eigenvector pair.

For directed graphs this is "left" eigenvector centrality which corresponds to the in-edges in the graph. For out-edges eigenvector centrality first reverse the graph with `G.reverse()`.

References

[1], [2]

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']
```

katz_centrality

katz_centrality(*G*, *alpha=0.1*, *beta=1.0*, *max_iter=1000*, *tol=1e-06*, *nstart=None*, *normalized=True*, *weight=None*)

Compute the Katz centrality for the nodes of the graph G.

Katz centrality computes the centrality for a node based on the centrality of its neighbors. It is a generalization of the eigenvector centrality. The Katz centrality for node i is

$$x_i = \alpha \sum_j A_{ij} x_j + \beta,$$

where A is the adjacency matrix of graph G with eigenvalues λ.

The parameter β controls the initial centrality and

$$\alpha < \frac{1}{\lambda_{\max}}.$$

Katz centrality computes the relative influence of a node within a network by measuring the number of the immediate neighbors (first degree nodes) and also all other nodes in the network that connect to the node under consideration through these immediate neighbors.

Extra weight can be provided to immediate neighbors through the parameter β. Connections made with distant neighbors are, however, penalized by an attenuation factor α which should be strictly less than the inverse largest eigenvalue of the adjacency matrix in order for the Katz centrality to be computed correctly. More information is provided in [1].

Parameters

G
[graph] A NetworkX graph.

alpha
[float] Attenuation factor

beta
[scalar or dictionary, optional (default=1.0)] Weight attributed to the immediate neighborhood. If not a scalar, the dictionary must have an value for every node.

max_iter
[integer, optional (default=1000)] Maximum number of iterations in power method.

tol
[float, optional (default=1.0e-6)] Error tolerance used to check convergence in power method iteration.

nstart
[dictionary, optional] Starting value of Katz iteration for each node.

normalized
[bool, optional (default=True)] If True normalize the resulting 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. In this measure the weight is interpreted as the connection strength.

Returns

nodes
[dictionary] Dictionary of nodes with Katz centrality as the value.

Raises

NetworkXError
If the parameter `beta` is not a scalar but lacks a value for at least one node

PowerIterationFailedConvergence
If the algorithm fails to converge to the specified tolerance within the specified number of iterations of the power iteration method.

See also:

katz_centrality_numpy
eigenvector_centrality
eigenvector_centrality_numpy
pagerank
hits

Notes

Katz centrality was introduced by [2].

This algorithm it uses the power method to find the eigenvector corresponding to the largest eigenvalue of the adjacency matrix of `G`. The parameter `alpha` should be strictly less than the inverse of largest eigenvalue of the adjacency matrix for the algorithm to converge. You can use `max(nx.adjacency_spectrum(G))` to get $\lambda_{\max}$ the largest eigenvalue of the adjacency matrix. The iteration will stop after `max_iter` iterations or an error tolerance of `number_of_nodes(G) * tol` has been reached.

When $\alpha = 1/\lambda_{\max}$ and $\beta = 0$, Katz centrality is the same as eigenvector centrality.

For directed graphs this finds "left" eigenvectors which corresponds to the in-edges in the graph. For out-edges Katz centrality first reverse the graph with `G.reverse()`.

References

[1], [2]

Examples

```
>>> import math
>>> G = nx.path_graph(4)
>>> phi = (1 + math.sqrt(5)) / 2.0  # largest eigenvalue of adj matrix
>>> centrality = nx.katz_centrality(G, 1 / phi - 0.01)
>>> for n, c in sorted(centrality.items()):
...     print(f"{n} {c:.2f}")
0 0.37
1 0.60
2 0.60
3 0.37
```

katz_centrality_numpy

katz_centrality_numpy(*G*, *alpha=0.1*, *beta=1.0*, *normalized=True*, *weight=None*)

Compute the Katz centrality for the graph G.

Katz centrality computes the centrality for a node based on the centrality of its neighbors. It is a generalization of the eigenvector centrality. The Katz centrality for node i is

$$x_i = \alpha \sum_j A_{ij} x_j + \beta,$$

where A is the adjacency matrix of graph G with eigenvalues λ.

The parameter β controls the initial centrality and

$$\alpha < \frac{1}{\lambda_{\max}}.$$

Katz centrality computes the relative influence of a node within a network by measuring the number of the immediate neighbors (first degree nodes) and also all other nodes in the network that connect to the node under consideration through these immediate neighbors.

Extra weight can be provided to immediate neighbors through the parameter β. Connections made with distant neighbors are, however, penalized by an attenuation factor α which should be strictly less than the inverse largest

eigenvalue of the adjacency matrix in order for the Katz centrality to be computed correctly. More information is provided in [1].

Parameters

G
[graph] A NetworkX graph

alpha
[float] Attenuation factor

beta
[scalar or dictionary, optional (default=1.0)] Weight attributed to the immediate neighborhood. If not a scalar the dictionary must have an value for every node.

normalized
[bool] If True normalize the resulting values.

weight
[None or string, optional] If None, all edge weights are considered equal. Otherwise holds the name of the edge attribute used as weight. In this measure the weight is interpreted as the connection strength.

Returns

nodes
[dictionary] Dictionary of nodes with Katz centrality as the value.

Raises

NetworkXError
If the parameter `beta` is not a scalar but lacks a value for at least one node

See also:

katz_centrality
eigenvector_centrality_numpy
eigenvector_centrality
pagerank
hits

Notes

Katz centrality was introduced by [2].

This algorithm uses a direct linear solver to solve the above equation. The parameter `alpha` should be strictly less than the inverse of largest eigenvalue of the adjacency matrix for there to be a solution. You can use `max(nx.adjacency_spectrum(G))` to get $\lambda_{\max}$ the largest eigenvalue of the adjacency matrix.

When $\alpha = 1/\lambda_{\max}$ and $\beta = 0$, Katz centrality is the same as eigenvector centrality.

For directed graphs this finds "left" eigenvectors which corresponds to the in-edges in the graph. For out-edges Katz centrality first reverse the graph with `G.reverse()`.

References

[1], [2]

Examples

```
>>> import math
>>> G = nx.path_graph(4)
>>> phi = (1 + math.sqrt(5)) / 2.0  # largest eigenvalue of adj matrix
>>> centrality = nx.katz_centrality_numpy(G, 1 / phi)
>>> for n, c in sorted(centrality.items()):
...     print(f"{n} {c:.2f}")
0 0.37
1 0.60
2 0.60
3 0.37
```

3.7.3 Closeness

closeness_centrality(G[, u, distance, ...])	Compute closeness centrality for nodes.
incremental_closeness_centrality(G, edge[, ...])	Incremental closeness centrality for nodes.

closeness_centrality

closeness_centrality(*G*, *u=None*, *distance=None*, *wf_improved=True*)

Compute closeness centrality for nodes.

Closeness centrality [1] of a node `u` is the reciprocal of the average shortest path distance to `u` over all `n-1` reachable nodes.

$$C(u) = \frac{n-1}{\sum_{v=1}^{n-1} d(v,u)},$$

where `d(v, u)` is the shortest-path distance between `v` and `u`, and `n-1` is the number of nodes reachable from `u`. Notice that the closeness distance function computes the incoming distance to `u` for directed graphs. To use outward distance, act on `G.reverse()`.

Notice that higher values of closeness indicate higher centrality.

Wasserman and Faust propose an improved formula for graphs with more than one connected component. The result is "a ratio of the fraction of actors in the group who are reachable, to the average distance" from the reachable actors [2]. You might think this scale factor is inverted but it is not. As is, nodes from small components receive a smaller closeness value. Letting `N` denote the number of nodes in the graph,

$$C_{WF}(u) = \frac{n-1}{N-1}\frac{n-1}{\sum_{v=1}^{n-1} d(v,u)},$$

Parameters

G

[graph] A NetworkX graph

u
: [node, optional] Return only the value for node u

distance
: [edge attribute key, optional (default=None)] Use the specified edge attribute as the edge distance in shortest path calculations. If `None` (the default) all edges have a distance of 1. Absent edge attributes are assigned a distance of 1. Note that no check is performed to ensure that edges have the provided attribute.

wf_improved
: [bool, optional (default=True)] If True, scale by the fraction of nodes reachable. This gives the Wasserman and Faust improved formula. For single component graphs it is the same as the original formula.

Returns

nodes
: [dictionary] Dictionary of nodes with closeness centrality as the value.

See also:

betweenness_centrality, ***load_centrality***, ***eigenvector_centrality***
degree_centrality, ***incremental_closeness_centrality***

Notes

The closeness centrality is normalized to `(n-1)/(|G|-1)` where `n` is the number of nodes in the connected part of graph containing the node. If the graph is not completely connected, this algorithm computes the closeness centrality for each connected part separately scaled by that parts size.

If the 'distance' keyword is set to an edge attribute key then the shortest-path length will be computed using Dijkstra's algorithm with that edge attribute as the edge weight.

The closeness centrality uses *inward* distance to a node, not outward. If you want to use outword distances apply the function to `G.reverse()`

In NetworkX 2.2 and earlier a bug caused Dijkstra's algorithm to use the outward distance rather than the inward distance. If you use a 'distance' keyword and a DiGraph, your results will change between v2.2 and v2.3.

References

[1], [2]

Examples

```
>>> G = nx.Graph([(0, 1), (0, 2), (0, 3), (1, 2), (1, 3)])
>>> nx.closeness_centrality(G)
{0: 1.0, 1: 1.0, 2: 0.75, 3: 0.75}
```

incremental_closeness_centrality

incremental_closeness_centrality(*G*, *edge*, *prev_cc=None*, *insertion=True*, *wf_improved=True*)

Incremental closeness centrality for nodes.

Compute closeness centrality for nodes using level-based work filtering as described in Incremental Algorithms for Closeness Centrality by Sariyuce et al.

Level-based work filtering detects unnecessary updates to the closeness centrality and filters them out.

— From "Incremental Algorithms for Closeness Centrality":

Theorem 1: Let $G = (V, E)$ be a graph and u and v be two vertices in V such that there is no edge (u, v) in E. Let $G' = (V, E \cup uv)$ Then $cc[s] = cc'[s]$ if and only if $|dG(s, u) - dG(s, v)| \leq 1$.

Where $dG(u, v)$ denotes the length of the shortest path between two vertices u, v in a graph G, cc[s] is the closeness centrality for a vertex s in V, and cc'[s] is the closeness centrality for a vertex s in V, with the (u, v) edge added. —

We use Theorem 1 to filter out updates when adding or removing an edge. When adding an edge (u, v), we compute the shortest path lengths from all other nodes to u and to v before the node is added. When removing an edge, we compute the shortest path lengths after the edge is removed. Then we apply Theorem 1 to use previously computed closeness centrality for nodes where $|dG(s, u) - dG(s, v)| \leq 1$. This works only for undirected, unweighted graphs; the distance argument is not supported.

Closeness centrality [1] of a node `u` is the reciprocal of the sum of the shortest path distances from `u` to all `n-1` other nodes. Since the sum of distances depends on the number of nodes in the graph, closeness is normalized by the sum of minimum possible distances `n-1`.

$$C(u) = \frac{n-1}{\sum_{v=1}^{n-1} d(v, u)},$$

where `d(v, u)` is the shortest-path distance between `v` and `u`, and `n` is the number of nodes in the graph.

Notice that higher values of closeness indicate higher centrality.

Parameters

G
[graph] A NetworkX graph

edge
[tuple] The modified edge (u, v) in the graph.

prev_cc
[dictionary] The previous closeness centrality for all nodes in the graph.

insertion
[bool, optional] If True (default) the edge was inserted, otherwise it was deleted from the graph.

wf_improved
[bool, optional (default=True)] If True, scale by the fraction of nodes reachable. This gives the Wasserman and Faust improved formula. For single component graphs it is the same as the original formula.

Returns

nodes
[dictionary] Dictionary of nodes with closeness centrality as the value.

See also:

betweenness_centrality, *load_centrality*, *eigenvector_centrality*
degree_centrality, *closeness_centrality*

Notes

The closeness centrality is normalized to `(n-1)/(|G|-1)` where `n` is the number of nodes in the connected part of graph containing the node. If the graph is not completely connected, this algorithm computes the closeness centrality for each connected part separately.

References

[1], [2]

3.7.4 Current Flow Closeness

current_flow_closeness_centrality(G[, ...])	Compute current-flow closeness centrality for nodes.
information_centrality(G[, weight, dtype, ...])	Compute current-flow closeness centrality for nodes.

current_flow_closeness_centrality

current_flow_closeness_centrality(*G*, *weight=None*, *dtype=<class 'float'>*, *solver='lu'*)

Compute current-flow closeness centrality for nodes.

Current-flow closeness centrality is variant of closeness centrality based on effective resistance between nodes in a network. This metric is also known as information centrality.

Parameters

G
[graph] A NetworkX graph.

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 reflects the capacity or the strength of the edge.

dtype: data type (default=float)
Default data type for internal matrices. Set to np.float32 for lower memory consumption.

solver: string (default='lu')
Type of linear solver to use for computing the flow matrix. Options are "full" (uses most memory), "lu" (recommended), and "cg" (uses least memory).

Returns

nodes
[dictionary] Dictionary of nodes with current flow closeness centrality as the value.

See also:

closeness_centrality

Notes

The algorithm is from Brandes [1].

See also [2] for the original definition of information centrality.

References

[1], [2]

information_centrality

information_centrality(*G*, *weight=None*, *dtype=<class 'float'>*, *solver='lu'*)

Compute current-flow closeness centrality for nodes.

Current-flow closeness centrality is variant of closeness centrality based on effective resistance between nodes in a network. This metric is also known as information centrality.

Parameters

G
[graph] A NetworkX graph.

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 reflects the capacity or the strength of the edge.

dtype: data type (default=float)
Default data type for internal matrices. Set to np.float32 for lower memory consumption.

solver: string (default='lu')
Type of linear solver to use for computing the flow matrix. Options are "full" (uses most memory), "lu" (recommended), and "cg" (uses least memory).

Returns

nodes
[dictionary] Dictionary of nodes with current flow closeness centrality as the value.

See also:

closeness_centrality

Notes

The algorithm is from Brandes [1].

See also [2] for the original definition of information centrality.

References

[1], [2]

3.7.5 (Shortest Path) Betweenness

betweenness_centrality(G[, k, normalized, ...])	Compute the shortest-path betweenness centrality for nodes.
betweenness_centrality_subset(G, sources, ...)	Compute betweenness centrality for a subset of nodes.
edge_betweenness_centrality(G[, k, ...])	Compute betweenness centrality for edges.
edge_betweenness_centrality_subset(G, ...[, ...])	Compute betweenness centrality for edges for a subset of nodes.

betweenness_centrality

betweenness_centrality(*G*, *k=None*, *normalized=True*, *weight=None*, *endpoints=False*, *seed=None*)

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

$$c_B(v) = \sum_{s,t \in V} \frac{\sigma(s,t|v)}{\sigma(s,t)}$$

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 *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 to count undirected paths. But if the source and target subsets differ – for example, if sources is $\{0\}$ and targets is $\{1\}$, then we are only counting the paths in one direction. They are undirected paths but we are counting them in a directed way. To count them as undirected paths, each should count as half a path.

References

[1], [2], [3], [4]

betweenness_centrality_subset

betweenness_centrality_subset(*G*, *sources*, *targets*, *normalized=False*, *weight=None*)

Compute betweenness centrality for a subset of nodes.

$$c_B(v) = \sum_{s \in S, t \in T} \frac{\sigma(s, t|v)}{\sigma(s, t)}$$

where S is the set of sources, T is the set of targets, $\sigma(s, t)$ 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.

sources: list of nodes
Nodes to use as sources for shortest paths in betweenness

targets: list of nodes
Nodes to use as targets for shortest paths in betweenness

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.

Returns

nodes
[dictionary] Dictionary of nodes with betweenness centrality as the value.

See also:

edge_betweenness_centrality
load_centrality

Notes

The basic algorithm is from [1].

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 normalization might seem a little strange but it is designed to make betweenness_centrality(G) be the same as betweenness_centrality_subset(G,sources=G.nodes(),targets=G.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 to count undirected paths. But if the source and target subsets differ – for example, if sources is $\{0\}$ and targets is $\{1\}$, then we are only counting the paths in one direction. They are undirected paths but we are counting them in a directed way. To count them as undirected paths, each should count as half a path.

References

[1], [2]

edge_betweenness_centrality

edge_betweenness_centrality(*G*, *k=None*, *normalized=True*, *weight=None*, *seed=None*)

Compute betweenness centrality for edges.

Betweenness centrality of an edge e is the sum of the fraction of all-pairs shortest paths that pass through e

$$c_B(e) = \sum_{s,t \in V} \frac{\sigma(s,t|e)}{\sigma(s,t)}$$

where V is the set of nodes, $\sigma(s,t)$ is the number of shortest (s,t)-paths, and $\sigma(s,t|e)$ is the number of those paths passing through edge e [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 $2/(n(n-1))$ for graphs, and $1/(n(n-1))$ 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.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*. Note that this is only used if k is not None.

Returns

edges
[dictionary] Dictionary of edges with betweenness centrality as the value.

See also:

betweenness_centrality
edge_load

Notes

The algorithm is from Ulrik Brandes [1].

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], [2]

edge_betweenness_centrality_subset

edge_betweenness_centrality_subset(*G*, *sources*, *targets*, *normalized=False*, *weight=None*)

Compute betweenness centrality for edges for a subset of nodes.

$$c_B(v) = \sum_{s \in S, t \in T} \frac{\sigma(s,t|e)}{\sigma(s,t)}$$

where S is the set of sources, T is the set of targets, $\sigma(s,t)$ is the number of shortest (s,t)-paths, and $\sigma(s,t|e)$ is the number of those paths passing through edge e [2].

Parameters

G
[graph] A networkx graph.

sources: list of nodes
Nodes to use as sources for shortest paths in betweenness

targets: list of nodes
Nodes to use as targets for shortest paths in betweenness

normalized
[bool, optional] If True the betweenness values are normalized by `2/(n(n-1))` for graphs, and `1/(n(n-1))` 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.

Returns

edges
[dictionary] Dictionary of edges with Betweenness centrality as the value.

See also:

betweenness_centrality
edge_load

Notes

The basic algorithm is from [1].

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 normalization might seem a little strange but it is the same as in edge_betweenness_centrality() and is designed to make edge_betweenness_centrality(G) be the same as edge_betweenness_centrality_subset(G,sources=G.nodes(),targets=G.nodes()).

References

[1], [2]

3.7.6 Current Flow Betweenness

current_flow_betweenness_centrality(G[, ...])	Compute current-flow betweenness centrality for nodes.
edge_current_flow_betweenness_centrality(G)	Compute current-flow betweenness centrality for edges.
approximate_current_flow_betweenness_centrality(G)	Compute the approximate current-flow betweenness centrality for nodes.
current_flow_betweenness_centrality_subset(G, ...)	Compute current-flow betweenness centrality for subsets of nodes.
edge_current_flow_betweenness_centrality_subset(G, ...)	Compute current-flow betweenness centrality for edges using subsets of nodes.

current_flow_betweenness_centrality

current_flow_betweenness_centrality(*G*, *normalized=True*, *weight=None*, *dtype=<class 'float'>*, *solver='full'*)

Compute current-flow betweenness centrality for nodes.

Current-flow betweenness centrality uses an electrical current model for information spreading in contrast to betweenness centrality which uses shortest paths.

Current-flow betweenness centrality is also known as random-walk betweenness centrality [2].

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).

Returns

nodes
[dictionary] Dictionary of nodes with betweenness centrality as the value.

See also:

approximate_current_flow_betweenness_centrality
betweenness_centrality
edge_betweenness_centrality
edge_current_flow_betweenness_centrality

Notes

Current-flow betweenness can be computed in $O(I(n-1)+mn\log n)$ time [1], where $I(n-1)$ is the time needed to compute the inverse Laplacian. For a full matrix this is $O(n^3)$ but using sparse methods you can achieve $O(nm\sqrt{k})$ where k is the Laplacian matrix condition number.

The space required is $O(nw)$ where w is the width of the sparse Laplacian matrix. Worse case is $w=n$ for $O(n^2)$.

If the edges have a 'weight' attribute they will be used as weights in this algorithm. Unspecified weights are set to 1.

References

[1], [2]

edge_current_flow_betweenness_centrality

edge_current_flow_betweenness_centrality(*G*, *normalized=True*, *weight=None*, *dtype=<class 'float'>*, *solver='full'*)

Compute current-flow betweenness centrality for edges.

Current-flow betweenness centrality uses an electrical current model for information spreading in contrast to betweenness centrality which uses shortest paths.

Current-flow betweenness centrality is also known as random-walk betweenness centrality [2].

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 (default=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).

Returns

nodes
[dictionary] Dictionary of edge tuples with betweenness centrality as the value.

Raises

NetworkXError
The algorithm does not support DiGraphs. If the input graph is an instance of DiGraph class, NetworkXError is raised.

See also:

betweenness_centrality
edge_betweenness_centrality
current_flow_betweenness_centrality

Notes

Current-flow betweenness can be computed in $O(I(n-1)+mn\log n)$ time [1], where $I(n-1)$ is the time needed to compute the inverse Laplacian. For a full matrix this is $O(n^3)$ but using sparse methods you can achieve $O(nm\sqrt{k})$ where k is the Laplacian matrix condition number.

The space required is $O(nw)$ where w is the width of the sparse Laplacian matrix. Worse case is $w=n$ for $O(n^2)$.

If the edges have a 'weight' attribute they will be used as weights in this algorithm. Unspecified weights are set to 1.

References

[1], [2]

approximate_current_flow_betweenness_centrality

approximate_current_flow_betweenness_centrality(*G*, *normalized=True*, *weight=None*, *dtype=<class 'float'>*, *solver='full'*, *epsilon=0.5*, *kmax=10000*, *seed=None*)

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 *Randomness*.

Returns

nodes
[dictionary] Dictionary of nodes with betweenness centrality as the value.

See also:

current_flow_betweenness_centrality

Notes

The running time is $O((1/\epsilon^2)m\sqrt{k}\log n)$ and the space required is $O(m)$ for n nodes and m edges.

If the edges have a 'weight' attribute they will be used as weights in this algorithm. Unspecified weights are set to 1.

References

[1]

current_flow_betweenness_centrality_subset

current_flow_betweenness_centrality_subset(*G*, *sources*, *targets*, *normalized=True*, *weight=None*, *dtype=<class 'float'>*, *solver='lu'*)

Compute current-flow betweenness centrality for subsets of nodes.

Current-flow betweenness centrality uses an electrical current model for information spreading in contrast to betweenness centrality which uses shortest paths.

Current-flow betweenness centrality is also known as random-walk betweenness centrality [2].

Parameters

G
[graph] A NetworkX graph

sources: list of nodes
Nodes to use as sources for current

targets: list of nodes
Nodes to use as sinks for current

normalized
[bool, optional (default=True)] If True the betweenness values are normalized by b=b/(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='lu')
Type of linear solver to use for computing the flow matrix. Options are "full" (uses most memory), "lu" (recommended), and "cg" (uses least memory).

Returns

nodes
[dictionary] Dictionary of nodes with betweenness centrality as the value.

See also:

approximate_current_flow_betweenness_centrality
betweenness_centrality
edge_betweenness_centrality
edge_current_flow_betweenness_centrality

Notes

Current-flow betweenness can be computed in $O(I(n-1)+mn\log n)$ time [1], where $I(n-1)$ is the time needed to compute the inverse Laplacian. For a full matrix this is $O(n^3)$ but using sparse methods you can achieve $O(nm\sqrt{k})$ where k is the Laplacian matrix condition number.

The space required is $O(nw)$ where w is the width of the sparse Laplacian matrix. Worse case is $w=n$ for $O(n^2)$.

If the edges have a 'weight' attribute they will be used as weights in this algorithm. Unspecified weights are set to 1.

References

[1], [2]

edge_current_flow_betweenness_centrality_subset

edge_current_flow_betweenness_centrality_subset(*G*, *sources*, *targets*, *normalized=True*, *weight=None*, *dtype=<class 'float'>*, *solver='lu'*)

Compute current-flow betweenness centrality for edges using subsets of nodes.

Current-flow betweenness centrality uses an electrical current model for information spreading in contrast to betweenness centrality which uses shortest paths.

Current-flow betweenness centrality is also known as random-walk betweenness centrality [2].

Parameters

G
[graph] A NetworkX graph

sources: list of nodes
Nodes to use as sources for current

targets: list of nodes
Nodes to use as sinks for current

normalized
[bool, optional (default=True)] If True the betweenness values are normalized by b=b/(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='lu')
Type of linear solver to use for computing the flow matrix. Options are "full" (uses most memory), "lu" (recommended), and "cg" (uses least memory).

Returns

nodes
[dict] Dictionary of edge tuples with betweenness centrality as the value.

See also:

betweenness_centrality
edge_betweenness_centrality
current_flow_betweenness_centrality

Notes

Current-flow betweenness can be computed in $O(I(n-1)+mn\log n)$ time [1], where $I(n-1)$ is the time needed to compute the inverse Laplacian. For a full matrix this is $O(n^3)$ but using sparse methods you can achieve $O(nm\sqrt{k})$ where k is the Laplacian matrix condition number.

The space required is $O(nw)$ where w is the width of the sparse Laplacian matrix. Worse case is $w=n$ for $O(n^2)$.

If the edges have a 'weight' attribute they will be used as weights in this algorithm. Unspecified weights are set to 1.

References

[1], [2]

3.7.7 Communicability Betweenness

communicability_betweenness_centrality(G)	Returns subgraph communicability for all pairs of nodes in G.

communicability_betweenness_centrality

communicability_betweenness_centrality(*G*)

Returns subgraph communicability for all pairs of nodes in G.

Communicability betweenness measure makes use of the number of walks connecting every pair of nodes as the basis of a betweenness centrality measure.

Parameters

G: graph

Returns

nodes
[dictionary] Dictionary of nodes with communicability betweenness as the value.

Raises

NetworkXError
If the graph is not undirected and simple.

Notes

Let `G=(V,E)` be a simple undirected graph with `n` nodes and `m` edges, and `A` denote the adjacency matrix of `G`.

Let `G(r)=(V,E(r))` be the graph resulting from removing all edges connected to node `r` but not the node itself.

The adjacency matrix for `G(r)` is `A+E(r)`, where `E(r)` has nonzeros only in row and column `r`.

The subraph betweenness of a node `r` is [1]

$$\omega_r = \frac{1}{C}\sum_p\sum_q \frac{G_{prq}}{G_{pq}}, p \neq q, q \neq r,$$

where `G_{prq}=(e^{A}_{pq} - (e^{A+E(r)})_{pq}` is the number of walks involving node r, `G_{pq}=(e^{A})_{pq}` is the number of closed walks starting at node `p` and ending at node `q`, and `C=(n-1)^{2}-(n-1)` is a normalization factor equal to the number of terms in the sum.

The resulting `omega_{r}` takes values between zero and one. The lower bound cannot be attained for a connected graph, and the upper bound is attained in the star graph.

References

[1]

Examples

```
>>> G = nx.Graph([(0, 1), (1, 2), (1, 5), (5, 4), (2, 4), (2, 3), (4, 3), (3, 6)])
>>> cbc = nx.communicability_betweenness_centrality(G)
>>> print([f"{node} {cbc[node]:0.2f}" for node in sorted(cbc)])
['0 0.03', '1 0.45', '2 0.51', '3 0.45', '4 0.40', '5 0.19', '6 0.03']
```

3.7.8 Group Centrality

group_betweenness_centrality(G, C[, ...])	Compute the group betweenness centrality for a group of nodes.
group_closeness_centrality(G, S[, weight])	Compute the group closeness centrality for a group of nodes.
group_degree_centrality(G, S)	Compute the group degree centrality for a group of nodes.
group_in_degree_centrality(G, S)	Compute the group in-degree centrality for a group of nodes.
group_out_degree_centrality(G, S)	Compute the group out-degree centrality for a group of nodes.
prominent_group(G, k[, weight, C, ...])	Find the prominent group of size k in graph G.

group_betweenness_centrality

group_betweenness_centrality(*G*, *C*, *normalized=True*, *weight=None*, *endpoints=False*)

Compute the group betweenness centrality for a group of nodes.

Group betweenness centrality of a group of nodes C is the sum of the fraction of all-pairs shortest paths that pass through any vertex in C

$$c_B(v) = \sum_{s,t \in V} \frac{\sigma(s, t|v)}{\sigma(s, t)}$$

where V is the set of nodes, $\sigma(s, t)$ is the number of shortest (s, t)-paths, and $\sigma(s, t|C)$ is the number of those paths passing through some node in group C. Note that (s, t) are not members of the group ($V - C$ is the set of nodes in V that are not in C).

Parameters

G
: [graph] A NetworkX graph.

C
: [list or set or list of lists or list of sets] A group or a list of groups containing nodes which belong to G, for which group betweenness centrality is to be calculated.

normalized
: [bool, optional (default=True)] If True, group betweenness is normalized by `1/((|V|-|C|)(|V|-|C|-1))` where `|V|` is the number of nodes in G and `|C|` is the number of nodes in C.

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.

endpoints
: [bool, optional (default=False)] If True include the endpoints in the shortest path counts.

Returns

betweenness
: [list of floats or float] If C is a single group then return a float. If C is a list with several groups then return a list of group betweenness centralities.

Raises

NodeNotFound
: If node(s) in C are not present in G.

See also:

betweenness_centrality

Notes

Group betweenness centrality is described in [1] and its importance discussed in [3]. The initial implementation of the algorithm is mentioned in [2]. This function uses an improved algorithm presented in [4].

The number of nodes in the group must be a maximum of n - 2 where `n` is the total number of nodes in the graph.

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 between "u" and "v" are counted as two possible paths (one each direction) while undirected paths between "u" and "v" are counted as one path. Said another way, the sum in the expression above is over all `s != t` for directed graphs and for `s < t` for undirected graphs.

References

[1], [2], [3], [4]

group_closeness_centrality

group_closeness_centrality(*G*, *S*, *weight=None*)

Compute the group closeness centrality for a group of nodes.

Group closeness centrality of a group of nodes S is a measure of how close the group is to the other nodes in the graph.

$$c_{close}(S) = \frac{|V - S|}{\sum_{v \in V - S} d_{S,v}}$$
$$d_{S,v} = min_{u \in S}(d_{u,v})$$

where V is the set of nodes, $d_{S,v}$ is the distance of the group S from v defined as above. ($V - S$ is the set of nodes in V that are not in S).

Parameters

G
[graph] A NetworkX graph.

S
[list or set] S is a group of nodes which belong to G, for which group closeness centrality is to be calculated.

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

closeness
[float] Group closeness centrality of the group S.

Raises

NodeNotFound
If node(s) in S are not present in G.

See also:

closeness_centrality

Notes

The measure was introduced in [1]. The formula implemented here is described in [2].

Higher values of closeness indicate greater centrality.

It is assumed that 1 / 0 is 0 (required in the case of directed graphs, or when a shortest path length is 0).

The number of nodes in the group must be a maximum of n - 1 where `n` is the total number of nodes in the graph.

For directed graphs, the incoming distance is utilized here. To use the outward distance, act on `G.reverse()`.

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], [2]

group_degree_centrality

group_degree_centrality(*G*, *S*)

Compute the group degree centrality for a group of nodes.

Group degree centrality of a group of nodes S is the fraction of non-group members connected to group members.

Parameters

G
[graph] A NetworkX graph.

S
[list or set] S is a group of nodes which belong to G, for which group degree centrality is to be calculated.

Returns

centrality
[float] Group degree centrality of the group S.

Raises

NetworkXError
If node(s) in S are not in G.

See also:

degree_centrality
group_in_degree_centrality
group_out_degree_centrality

Notes

The measure was introduced in [1].

The number of nodes in the group must be a maximum of n - 1 where `n` is the total number of nodes in the graph.

References

[1]

group_in_degree_centrality

group_in_degree_centrality(*G*, *S*)

Compute the group in-degree centrality for a group of nodes.

Group in-degree centrality of a group of nodes S is the fraction of non-group members connected to group members by incoming edges.

Parameters

G
[graph] A NetworkX graph.

S
[list or set] S is a group of nodes which belong to G, for which group in-degree centrality is to be calculated.

Returns

centrality
[float] Group in-degree centrality of the group S.

Raises

NetworkXNotImplemented
If G is undirected.

NodeNotFound
If node(s) in S are not in G.

See also:

degree_centrality
group_degree_centrality
group_out_degree_centrality

Notes

The number of nodes in the group must be a maximum of n - 1 where `n` is the total number of nodes in the graph.

`G.neighbors(i)` gives nodes with an outward edge from i, in a DiGraph, so for group in-degree centrality, the reverse graph is used.

group_out_degree_centrality

group_out_degree_centrality(*G*, *S*)

Compute the group out-degree centrality for a group of nodes.

Group out-degree centrality of a group of nodes S is the fraction of non-group members connected to group members by outgoing edges.

Parameters

G
[graph] A NetworkX graph.

S
[list or set] S is a group of nodes which belong to G, for which group in-degree centrality is to be calculated.

Returns

centrality
[float] Group out-degree centrality of the group S.

Raises

NetworkXNotImplemented
If G is undirected.

NodeNotFound
If node(s) in S are not in G.

See also:

degree_centrality
group_degree_centrality
group_in_degree_centrality

Notes

The number of nodes in the group must be a maximum of n - 1 where `n` is the total number of nodes in the graph.

`G.neighbors(i)` gives nodes with an outward edge from i, in a DiGraph, so for group out-degree centrality, the graph itself is used.

prominent_group

prominent_group(*G*, *k*, *weight=None*, *C=None*, *endpoints=False*, *normalized=True*, *greedy=False*)

Find the prominent group of size k in graph G. The prominence of the group is evaluated by the group betweenness centrality.

Group betweenness centrality of a group of nodes C is the sum of the fraction of all-pairs shortest paths that pass through any vertex in C

$$c_B(v) = \sum_{s,t \in V} \frac{\sigma(s, t|v)}{\sigma(s, t)}$$

where V is the set of nodes, $\sigma(s, t)$ is the number of shortest (s, t)-paths, and $\sigma(s, t|C)$ is the number of those paths passing through some node in group C. Note that (s, t) are not members of the group ($V - C$ is the set of nodes in V that are not in C).

Parameters

G
[graph] A NetworkX graph.

k
[int] The number of nodes in the group.

normalized
[bool, optional (default=True)] If True, group betweenness is normalized by `1/((|V|-|C|)(|V|-|C|-1))` where `|V|` is the number of nodes in G and `|C|` is the number of nodes in C.

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.

endpoints
[bool, optional (default=False)] If True include the endpoints in the shortest path counts.

C
[list or set, optional (default=None)] list of nodes which won't be candidates of the prominent group.

greedy
[bool, optional (default=False)] Using a naive greedy algorithm in order to find non-optimal prominent group. For scale free networks the results are negligibly below the optimal results.

Returns

max_GBC
[float] The group betweenness centrality of the prominent group.

max_group
[list] The list of nodes in the prominent group.

Raises

NodeNotFound
If node(s) in C are not present in G.

See also:

betweenness_centrality, *group_betweenness_centrality*

Notes

Group betweenness centrality is described in [1] and its importance discussed in [3]. The algorithm is described in [2] and is based on techniques mentioned in [4].

The number of nodes in the group must be a maximum of `n - 2` where `n` is the total number of nodes in the graph.

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 between "u" and "v" are counted as two possible paths (one each direction) while undirected paths between "u" and "v" are counted as one path. Said another way, the sum in the expression above is over all `s != t` for directed graphs and for `s < t` for undirected graphs.

References

[1], [2], [3], [4]

3.7.9 Load

load_centrality(G[, v, cutoff, normalized, ...])	Compute load centrality for nodes.
edge_load_centrality(G[, cutoff])	Compute edge load.

load_centrality

load_centrality(*G*, *v=None*, *cutoff=None*, *normalized=True*, *weight=None*)

Compute load centrality for nodes.

The load centrality of a node is the fraction of all shortest paths that pass through that node.

Parameters

G
[graph] A networkx graph.

normalized
[bool, optional (default=True)] If True the betweenness values are normalized by b=b/(n-1)(n-2) where n is the number of nodes in G.

weight
[None or string, optional (default=None)] If None, edge weights are ignored. 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.

cutoff
[bool, optional (default=None)] If specified, only consider paths of length <= cutoff.

Returns

nodes
[dictionary] Dictionary of nodes with centrality as the value.

See also:

betweenness_centrality

Notes

Load centrality is slightly different than betweenness. It was originally introduced by [2]. For this load algorithm see [1].

References

[1], [2]

edge_load_centrality

edge_load_centrality(*G*, *cutoff=False*)

Compute edge load.

WARNING: This concept of edge load has not been analysed or discussed outside of NetworkX that we know of. It is based loosely on load_centrality in the sense that it counts the number of shortest paths which cross each edge. This function is for demonstration and testing purposes.

Parameters

G
[graph] A networkx graph

cutoff
[bool, optional (default=False)] If specified, only consider paths of length <= cutoff.

Returns

A dict keyed by edge 2-tuple to the number of shortest paths which use that edge. Where more than one path is shortest the count is divided equally among paths.

3.7.10 Subgraph

subgraph_centrality(G)	Returns subgraph centrality for each node in G.
subgraph_centrality_exp(G)	Returns the subgraph centrality for each node of G.
estrada_index(G)	Returns the Estrada index of a the graph G.

subgraph_centrality

subgraph_centrality(*G*)

Returns subgraph centrality for each node in G.

Subgraph centrality of a node `n` is the sum of weighted closed walks of all lengths starting and ending at node `n`. The weights decrease with path length. Each closed walk is associated with a connected subgraph ([1]).

Parameters

G: graph

Returns

nodes
[dictionary] Dictionary of nodes with subgraph centrality as the value.

Raises

NetworkXError
If the graph is not undirected and simple.

See also:

subgraph_centrality_exp
Alternative algorithm of the subgraph centrality for each node of G.

Notes

This version of the algorithm computes eigenvalues and eigenvectors of the adjacency matrix.

Subgraph centrality of a node `u` in G can be found using a spectral decomposition of the adjacency matrix [1],

$$SC(u) = \sum_{j=1}^{N} (v_j^u)^2 e^{\lambda_j},$$

where `v_j` is an eigenvector of the adjacency matrix `A` of G corresponding to the eigenvalue `lambda_j`.

References

[1]

Examples

(Example from [1]) >>> G = nx.Graph(… [… (1, 2), … (1, 5), … (1, 8), … (2, 3), … (2, 8), … (3, 4), … (3, 6), … (4, 5), … (4, 7), … (5, 6), … (6, 7), … (7, 8), …] …) >>> sc = nx.subgraph_centrality(G) >>> print([f"{node} {sc[node]:0.2f}" for node in sorted(sc)]) ['1 3.90', '2 3.90', '3 3.64', '4 3.71', '5 3.64', '6 3.71', '7 3.64', '8 3.90']

subgraph_centrality_exp

subgraph_centrality_exp(*G*)
Returns the subgraph centrality for each node of G.

Subgraph centrality of a node `n` is the sum of weighted closed walks of all lengths starting and ending at node `n`. The weights decrease with path length. Each closed walk is associated with a connected subgraph ([1]).

Parameters

G: graph

Returns

nodes:dictionary
Dictionary of nodes with subgraph centrality as the value.

Raises

NetworkXError
If the graph is not undirected and simple.

See also:

subgraph_centrality
Alternative algorithm of the subgraph centrality for each node of G.

Notes

This version of the algorithm exponentiates the adjacency matrix.

The subgraph centrality of a node `u` in G can be found using the matrix exponential of the adjacency matrix of G [1],

$$SC(u) = (e^A)_{uu}.$$

References

[1]

Examples

(Example from [1]) >>> G = nx.Graph(… [… (1, 2), … (1, 5), … (1, 8), … (2, 3), … (2, 8), … (3, 4), … (3, 6), … (4, 5), … (4, 7), … (5, 6), … (6, 7), … (7, 8), …] …) >>> sc = nx.subgraph_centrality_exp(G) >>> print([f"{node} {sc[node]:0.2f}" for node in sorted(sc)]) ['1 3.90', '2 3.90', '3 3.64', '4 3.71', '5 3.64', '6 3.71', '7 3.64', '8 3.90']

estrada_index

estrada_index(*G*)

Returns the Estrada index of a the graph G.

The Estrada Index is a topological index of folding or 3D "compactness" ([1]).

Parameters

G: graph

Returns

estrada index: float

Raises

NetworkXError
If the graph is not undirected and simple.

Notes

Let `G=(V,E)` be a simple undirected graph with `n` nodes and let `lambda_{1}leqlambda_{2}leqcdotslambda_{n}` be a non-increasing ordering of the eigenvalues of its adjacency matrix `A`. The Estrada index is ([1], [2])

$$EE(G) = \sum_{j=1}^{n} e^{\lambda_j}.$$

References

[1], [2]

Examples

```
>>> G = nx.Graph([(0, 1), (1, 2), (1, 5), (5, 4), (2, 4), (2, 3), (4, 3), (3, 6)])
>>> ei = nx.estrada_index(G)
>>> print(f"{ei:0.5}")
20.55
```

3.7.11 Harmonic Centrality

harmonic_centrality(G[, nbunch, distance, ...])	Compute harmonic centrality for nodes.

harmonic_centrality

harmonic_centrality(*G*, *nbunch=None*, *distance=None*, *sources=None*)

Compute harmonic centrality for nodes.

Harmonic centrality [1] of a node `u` is the sum of the reciprocal of the shortest path distances from all other nodes to `u`

$$C(u) = \sum_{v \neq u} \frac{1}{d(v, u)}$$

where `d(v, u)` is the shortest-path distance between `v` and `u`.

If `sources` is given as an argument, the returned harmonic centrality values are calculated as the sum of the reciprocals of the shortest path distances from the nodes specified in `sources` to `u` instead of from all nodes to `u`.

Notice that higher values indicate higher centrality.

Parameters

G
[graph] A NetworkX graph

nbunch
[container (default: all nodes in G)] Container of nodes for which harmonic centrality values are calculated.

sources
[container (default: all nodes in G)] Container of nodes `v` over which reciprocal distances are computed. Nodes not in `G` are silently ignored.

distance
[edge attribute key, optional (default=None)] Use the specified edge attribute as the edge distance in shortest path calculations. If `None`, then each edge will have distance equal to 1.

Returns

nodes
[dictionary] Dictionary of nodes with harmonic centrality as the value.

See also:

betweenness_centrality, *load_centrality*, *eigenvector_centrality*
degree_centrality, *closeness_centrality*

Notes

If the 'distance' keyword is set to an edge attribute key then the shortest-path length will be computed using Dijkstra's algorithm with that edge attribute as the edge weight.

References

[1]

3.7.12 Dispersion

dispersion(G[, u, v, normalized, alpha, b, c])	Calculate dispersion between `u` and `v` in `G`.

dispersion

dispersion(*G*, *u=None*, *v=None*, *normalized=True*, *alpha=1.0*, *b=0.0*, *c=0.0*)

Calculate dispersion between `u` and `v` in `G`.

A link between two actors (`u` and `v`) has a high dispersion when their mutual ties (`s` and `t`) are not well connected with each other.

Parameters

G
[graph] A NetworkX graph.

u
[node, optional] The source for the dispersion score (e.g. ego node of the network).

v
[node, optional] The target of the dispersion score if specified.

normalized
[bool] If True (default) normalize by the embeddedness of the nodes (u and v).

alpha, b, c
[float] Parameters for the normalization procedure. When `normalized` is True, the dispersion value is normalized by:

```
result = ((dispersion + b) ** alpha) / (embeddedness + c)
```

as long as the denominator is nonzero.

Returns

nodes
[dictionary] If u (v) is specified, returns a dictionary of nodes with dispersion score for all "target" ("source") nodes. If neither u nor v is specified, returns a dictionary of dictionaries for all nodes 'u' in the graph with a dispersion score for each node 'v'.

Notes

This implementation follows Lars Backstrom and Jon Kleinberg [1]. Typical usage would be to run dispersion on the ego network G_u if u were specified. Running `dispersion()` with neither u nor v specified can take some time to complete.

References

[1]

3.7.13 Reaching

`local_reaching_centrality`(G, v[, paths, ...])	Returns the local reaching centrality of a node in a directed graph.
`global_reaching_centrality`(G[, weight, ...])	Returns the global reaching centrality of a directed graph.

local_reaching_centrality

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, `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`.

See also:

global_reaching_centrality

References

[1]

Examples

```
>>> G = nx.DiGraph()
>>> G.add_edges_from([(1, 2), (1, 3)])
>>> nx.local_reaching_centrality(G, 3)
0.0
>>> G.add_edge(3, 2)
>>> nx.local_reaching_centrality(G, 3)
0.5
```

global_reaching_centrality

global_reaching_centrality(*G*, *weight=None*, *normalized=True*)

Returns the global reaching centrality of a directed graph.

The *global reaching centrality* of a weighted directed graph is the average over all nodes of the difference between the local reaching centrality of the node and the greatest local reaching centrality of any node in the graph [1]. For more information on the local reaching centrality, see *local_reaching_centrality()*. Informally, the local reaching centrality is the proportion of the graph that is reachable from the neighbors of the node.

Parameters

G
[DiGraph] A networkx DiGraph.

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 global reaching centrality of the graph.

See also:

local_reaching_centrality

References

[1]

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
```

3.7.14 Percolation

percolation_centrality(G[, attribute, ...])	Compute the percolation centrality for nodes.

percolation_centrality

percolation_centrality(*G*, *attribute='percolation'*, *states=None*, *weight=None*)

Compute the percolation centrality for nodes.

Percolation centrality of a node v, at a given time, is defined as the proportion of 'percolated paths' that go through that node.

This measure quantifies relative impact of nodes based on their topological connectivity, as well as their percolation states.

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.

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 dependecies 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], [2]

3.7.15 Second Order Centrality

second_order_centrality(G)	Compute the second order centrality for nodes of G.

second_order_centrality

second_order_centrality(*G*)

Compute the second order centrality for nodes of G.

The second order centrality of a given node is the standard deviation of the return times to that node of a perpetual random walk on G:

Parameters

G
[graph] A NetworkX connected and undirected graph.

Returns

nodes
[dictionary] Dictionary keyed by node with second order centrality as the value.

Raises

NetworkXException
If the graph G is empty, non connected or has negative weights.

See also:

betweenness_centrality

Notes

Lower values of second order centrality indicate higher centrality.

The algorithm is from Kermarrec, Le Merrer, Sericola and Trédan [1].

This code implements the analytical version of the algorithm, i.e., there is no simulation of a random walk process involved. The random walk is here unbiased (corresponding to eq 6 of the paper [1]), thus the centrality values are the standard deviations for random walk return times on the transformed input graph G (equal in-degree at each nodes by adding self-loops).

Complexity of this implementation, made to run locally on a single machine, is O(n^3), with n the size of G, which makes it viable only for small graphs.

References

[1]

Examples

```
>>> G = nx.star_graph(10)
>>> soc = nx.second_order_centrality(G)
>>> print(sorted(soc.items(), key=lambda x: x[1])[0][0])  # pick first id
0
```

3.7.16 Trophic

trophic_levels(G[, weight])	Compute the trophic levels of nodes.
trophic_differences(G[, weight])	Compute the trophic differences of the edges of a directed graph.
trophic_incoherence_parameter(G[, weight, ...])	Compute the trophic incoherence parameter of a graph.

trophic_levels

trophic_levels(*G*, *weight='weight'*)

Compute the trophic levels of nodes.

The trophic level of a node i is

$$s_i = 1 + \frac{1}{k_i^{in}} \sum_j a_{ij} s_j$$

where k_i^{in} is the in-degree of i

$$k_i^{in} = \sum_j a_{ij}$$

and nodes with $k_i^{in} = 0$ have $s_i = 1$ by convention.

These are calculated using the method outlined in Levine [1].

Parameters

G
[DiGraph] A directed networkx graph

Returns

nodes
[dict] Dictionary of nodes with trophic level as the value.

References

[1]

trophic_differences

trophic_differences(*G*, *weight='weight'*)

Compute the trophic differences of the edges of a directed graph.

The trophic difference x_ij for each edge is defined in Johnson et al. [1] as:

$$x_ij = s_j - s_i$$

Where s_i is the trophic level of node i.

Parameters

G
[DiGraph] A directed networkx graph

Returns

diffs
[dict] Dictionary of edges with trophic differences as the value.

References

[1]

trophic_incoherence_parameter

trophic_incoherence_parameter(*G*, *weight='weight'*, *cannibalism=False*)

Compute the trophic incoherence parameter of a graph.

Trophic coherence is defined as the homogeneity of the distribution of trophic distances: the more similar, the more coherent. This is measured by the standard deviation of the trophic differences and referred to as the trophic incoherence parameter q by [1].

Parameters

G
[DiGraph] A directed networkx graph

cannibalism: Boolean
If set to False, self edges are not considered in the calculation

Returns

trophic_incoherence_parameter
[float] The trophic coherence of a graph

References

[1]

3.7.17 VoteRank

voterank(G[, number_of_nodes])	Select a list of influential nodes in a graph using VoteRank algorithm

voterank

voterank(*G*, *number_of_nodes=None*)

Select a list of influential nodes in a graph using VoteRank algorithm

VoteRank [1] computes a ranking of the nodes in a graph G based on a voting scheme. With VoteRank, all nodes vote for each of its in-neighbours and the node with the highest votes is elected iteratively. The voting ability of out-neighbors of elected nodes is decreased in subsequent turns.

Parameters

G
[graph] A NetworkX graph.

number_of_nodes
[integer, optional] Number of ranked nodes to extract (default all nodes).

Returns

voterank
[list] Ordered list of computed seeds. Only nodes with positive number of votes are returned.

Notes

Each edge is treated independently in case of multigraphs.

References

[1]

Examples

```
>>> G = nx.Graph([(0, 1), (0, 2), (0, 3), (1, 4)])
>>> nx.voterank(G)
[0, 1]
```

The algorithm can be used both for undirected and directed graphs. However, the directed version is different in two ways: (i) nodes only vote for their in-neighbors and (ii) only the voting ability of elected node and its out-neighbors are updated:

```
>>> G = nx.DiGraph([(0, 1), (2, 1), (2, 3), (3, 4)])
>>> nx.voterank(G)
[2, 3]
```

3.8 Chains

Functions for finding chains in a graph.

chain_decomposition(G[, root])	Returns the chain decomposition of a graph.

3.8.1 chain_decomposition

chain_decomposition(*G*, *root=None*)

Returns the chain decomposition of a graph.

The *chain decomposition* of a graph with respect a depth-first search tree is a set of cycles or paths derived from the set of fundamental cycles of the tree in the following manner. Consider each fundamental cycle with respect to the given tree, represented as a list of edges beginning with the nontree edge oriented away from the root of the tree. For each fundamental cycle, if it overlaps with any previous fundamental cycle, just take the initial non-overlapping segment, which is a path instead of a cycle. Each cycle or path is called a *chain*. For more information, see [1].

Parameters

G
[undirected graph]

root
[node (optional)] A node in the graph `G`. If specified, only the chain decomposition for the connected component containing this node will be returned. This node indicates the root of the depth-first search tree.

Yields

chain
[list] A list of edges representing a chain. There is no guarantee on the orientation of the edges in each chain (for example, if a chain includes the edge joining nodes 1 and 2, the chain may include either (1, 2) or (2, 1)).

Raises

NodeNotFound
If `root` is not in the graph `G`.

Notes

The worst-case running time of this implementation is linear in the number of nodes and number of edges [1].

References

[1]

Examples

```
>>> G = nx.Graph([(0, 1), (1, 4), (3, 4), (3, 5), (4, 5)])
>>> list(nx.chain_decomposition(G))
[[(4, 5), (5, 3), (3, 4)]]
```

3.9 Chordal

Algorithms for chordal graphs.

A graph is chordal if every cycle of length at least 4 has a chord (an edge joining two nodes not adjacent in the cycle). https://en.wikipedia.org/wiki/Chordal_graph

is_chordal(G)	Checks whether G is a chordal graph.
chordal_graph_cliques(G)	Returns all maximal cliques of a chordal graph.
chordal_graph_treewidth(G)	Returns the treewidth of the chordal graph G.
complete_to_chordal_graph(G)	Return a copy of G completed to a chordal graph
find_induced_nodes(G, s, t[, treewidth_bound])	Returns the set of induced nodes in the path from s to t.

3.9.1 is_chordal

is_chordal(*G*)

Checks whether G is a chordal graph.

A graph is chordal if every cycle of length at least 4 has a chord (an edge joining two nodes not adjacent in the cycle).

Parameters

G
[graph] A NetworkX graph.

Returns

chordal
[bool] True if G is a chordal graph and False otherwise.

Raises

NetworkXNotImplemented
The algorithm does not support DiGraph, MultiGraph and MultiDiGraph.

Notes

The routine tries to go through every node following maximum cardinality search. It returns False when it finds that the separator for any node is not a clique. Based on the algorithms in [1].

References

[1]

Examples

```
>>> e = [
...     (1, 2),
...     (1, 3),
...     (2, 3),
...     (2, 4),
...     (3, 4),
...     (3, 5),
...     (3, 6),
...     (4, 5),
...     (4, 6),
...     (5, 6),
... ]
>>> G = nx.Graph(e)
>>> nx.is_chordal(G)
True
```

3.9.2 chordal_graph_cliques

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 `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)
>>> G.add_node(9)
>>> cliques = [c for c in chordal_graph_cliques(G)]
>>> cliques[0]
frozenset({1, 2, 3})
```

3.9.3 chordal_graph_treewidth

chordal_graph_treewidth(*G*)

Returns the treewidth of the chordal graph G.

Parameters

G
[graph] A NetworkX graph

Returns

treewidth
[int] The size of the largest clique in the graph minus one.

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 `NetworkXError` is raised.

References

[1]

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)
>>> G.add_node(9)
>>> nx.chordal_graph_treewidth(G)
3
```

3.9.4 complete_to_chordal_graph

complete_to_chordal_graph(G)

Return a copy of G completed to a chordal graph

Adds edges to a copy of G to create a chordal graph. A graph G=(V,E) is called chordal if for each cycle with length bigger than 3, there exist two non-adjacent nodes connected by an edge (called a chord).

Parameters

G
[NetworkX graph] Undirected graph

Returns

H
[NetworkX graph] The chordal enhancement of G

alpha
[Dictionary] The elimination ordering of nodes of G

Notes

There are different approaches to calculate the chordal enhancement of a graph. The algorithm used here is called MCS-M and gives at least minimal (local) triangulation of graph. Note that this triangulation is not necessarily a global minimum.

https://en.wikipedia.org/wiki/Chordal_graph

References

[1]

Examples

```
>>> from networkx.algorithms.chordal import complete_to_chordal_graph
>>> G = nx.wheel_graph(10)
>>> H, alpha = complete_to_chordal_graph(G)
```

3.9.5 find_induced_nodes

find_induced_nodes(*G*, *s*, *t*, *treewidth_bound=9223372036854775807*)

Returns the set of induced nodes in the path from s to t.

Parameters

G
[graph] A chordal NetworkX graph

s
[node] Source node to look for induced nodes

t
[node] Destination node to look for induced nodes

treewidth_bound: float
Maximum treewidth acceptable for the graph H. The search for induced nodes will end as soon as the treewidth_bound is exceeded.

Returns

induced_nodes
[Set of nodes] The set of induced nodes in the path from s to t in G

Raises

NetworkXError
The algorithm does not support DiGraph, MultiGraph and MultiDiGraph. If the input graph is an instance of one of these classes, a `NetworkXError` is raised. The algorithm can only be applied to chordal graphs. If the input graph is found to be non-chordal, a `NetworkXError` is raised.

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.

References

[1]

Examples

```
>>> G = nx.Graph()
>>> G = nx.generators.classic.path_graph(10)
>>> induced_nodes = nx.find_induced_nodes(G, 1, 9, 2)
>>> sorted(induced_nodes)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

3.10 Clique

Functions for finding and manipulating cliques.

Finding the largest clique in a graph is NP-complete problem, so most of these algorithms have an exponential running time; for more information, see the Wikipedia article on the clique problem [1].

enumerate_all_cliques(G)	Returns all cliques in an undirected graph.
find_cliques(G[, nodes])	Returns all maximal cliques in an undirected graph.
find_cliques_recursive(G[, nodes])	Returns all maximal cliques in a graph.
make_max_clique_graph(G[, create_using])	Returns the maximal clique graph of the given graph.
make_clique_bipartite(G[, fpos, ...])	Returns the bipartite clique graph corresponding to G.
graph_clique_number(G[, cliques])	Returns the clique number of the graph.
graph_number_of_cliques(G[, cliques])	Returns the number of maximal cliques in the graph.
node_clique_number(G[, nodes, cliques, ...])	Returns the size of the largest maximal clique containing each given node.
number_of_cliques(G[, nodes, cliques])	Returns the number of maximal cliques for each node.
cliques_containing_node(G[, nodes, cliques])	Returns a list of cliques containing the given node.
max_weight_clique(G[, weight])	Find a maximum weight clique in G.

3.10.1 enumerate_all_cliques

enumerate_all_cliques(*G*)

Returns all cliques in an undirected graph.

This function returns an iterator over cliques, each of which is a list of nodes. The iteration is ordered by cardinality of the cliques: first all cliques of size one, then all cliques of size two, etc.

Parameters

G
[NetworkX graph] An undirected graph.

Returns

iterator
An iterator over cliques, each of which is a list of nodes in G. The cliques are ordered according to size.

Notes

To obtain a list of all cliques, use `list(enumerate_all_cliques(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 (for example, when the graph is the complete graph). This function avoids storing all cliques in memory by only keeping current candidate node lists in memory during its search.

The implementation is adapted from the algorithm by Zhang, et al. (2005) [1] to output all cliques discovered.

This algorithm ignores self-loops and parallel edges, since cliques are not conventionally defined with such edges.

References

[1]

3.10.2 find_cliques

find_cliques(*G*, *nodes=None*)

Returns all maximal cliques in an undirected graph.

For each node *n*, a *maximal clique for n* is a largest complete subgraph containing *n*. 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 an iterative implementation, so should not suffer from recursion depth issues.

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] An undirected 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 returned. The order of cliques is arbitrary.

Raises

ValueError
If `nodes` is not a clique.

See also:

find_cliques_recursive
A recursive version of the same algorithm.

Notes

To obtain a list of all maximal cliques, use `list(find_cliques(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]. It essentially unrolls the recursion used in the references to avoid issues of recursion stack depth (for a recursive implementation, see *find_cliques_recursive()*).

This algorithm ignores self-loops and parallel edges, since cliques are not conventionally defined with such edges.

References

[1], [2], [3]

3.10.3 find_cliques_recursive

find_cliques_recursive(*G*, *nodes=None*)

Returns all maximal cliques in a graph.

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 *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.

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 *find_cliques()*.

This algorithm ignores self-loops and parallel edges, since cliques are not conventionally defined with such edges.

References

[1], [2], [3]

3.10.4 make_max_clique_graph

make_max_clique_graph(*G*, *create_using=None*)

Returns the maximal clique graph of the given graph.

The nodes of the maximal clique graph of `G` are the cliques of `G` and an edge joins two cliques if the cliques are not disjoint.

Parameters

G
[NetworkX graph]

create_using
[NetworkX graph constructor, optional (default=nx.Graph)] Graph type to create. If graph instance, then cleared before populated.

Returns

NetworkX graph
A graph whose nodes are the cliques of `G` and whose edges join two cliques if they are not disjoint.

Notes

This function behaves like the following code:

```
import networkx as nx
G = nx.make_clique_bipartite(G)
cliques = [v for v in G.nodes() if G.nodes[v]['bipartite'] == 0]
G = nx.bipartite.projected_graph(G, cliques)
G = nx.relabel_nodes(G, {-v: v - 1 for v in G})
```

It should be faster, though, since it skips all the intermediate steps.

3.10.5 make_clique_bipartite

make_clique_bipartite(*G*, *fpos=None*, *create_using=None*, *name=None*)

Returns the bipartite clique graph corresponding to `G`.

In the returned bipartite graph, the "bottom" nodes are the nodes of `G` and the "top" nodes represent the maximal cliques of `G`. There is an edge from node *v* to clique *C* in the returned graph if and only if *v* is an element of *C*.

Parameters

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.

3.10.6 graph_clique_number

graph_clique_number(*G*, *cliques=None*)

Returns the clique number of the graph.

The *clique number* of a graph is the size of the largest clique in the graph.

Parameters

G
[NetworkX graph] An undirected graph.

cliques
[list] A list of cliques, each of which is itself a list of nodes. If not specified, the list of all cliques will be computed, as by *find_cliques()*.

Returns

int
The size of the largest clique in `G`.

Notes

You should provide `cliques` if you have already computed the list of maximal cliques, in order to avoid an exponential time search for maximal cliques.

3.10.7 graph_number_of_cliques

graph_number_of_cliques(*G*, *cliques=None*)

Returns the number of maximal cliques in the graph.

Parameters

G
[NetworkX graph] An undirected graph.

cliques
[list] A list of cliques, each of which is itself a list of nodes. If not specified, the list of all cliques will be computed, as by *find_cliques()*.

Returns

int
The number of maximal cliques in `G`.

Notes

You should provide `cliques` if you have already computed the list of maximal cliques, in order to avoid an exponential time search for maximal cliques.

3.10.8 node_clique_number

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 *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`.

3.10.9 number_of_cliques

number_of_cliques(*G*, *nodes=None*, *cliques=None*)

Returns the number of maximal cliques for each node.

Returns a single or list depending on input nodes. Optional list of cliques can be input if already computed.

3.10.10 cliques_containing_node

cliques_containing_node(*G*, *nodes=None*, *cliques=None*)

Returns a list of cliques containing the given node.

Returns a single list or list of lists depending on input nodes. Optional list of cliques can be input if already computed.

3.10.11 max_weight_clique

max_weight_clique(*G*, *weight='weight'*)

Find a maximum weight clique in G.

A *clique* in a graph is a set of nodes such that every two distinct nodes are adjacent. The *weight* of a clique is the sum of the weights of its nodes. A *maximum weight clique* of graph G is a clique C in G such that no clique in G has weight greater than the weight of C.

Parameters

G
[NetworkX graph] Undirected graph

weight
[string or None, optional (default='weight')] The node attribute that holds the integer value used as a weight. If None, then each node has weight 1.

Returns

clique
[list] the nodes of a maximum weight clique

weight
[int] the weight of a maximum weight clique

Notes

The implementation is recursive, and therefore it may run into recursion depth issues if G contains a clique whose number of nodes is close to the recursion depth limit.

At each search node, the algorithm greedily constructs a weighted independent set cover of part of the graph in order to find a small set of nodes on which to branch. The algorithm is very similar to the algorithm of Tavares et al. [1], other than the fact that the NetworkX version does not use bitsets. This style of algorithm for maximum weight clique (and maximum weight independent set, which is the same problem but on the complement graph) has a decades-long history. See Algorithm B of Warren and Hicks [2] and the references in that paper.

References

[1], [2]

3.11 Clustering

Algorithms to characterize the number of triangles in a graph.

triangles(G[, nodes])	Compute the number of triangles.
transitivity(G)	Compute graph transitivity, the fraction of all possible triangles present in G.
clustering(G[, nodes, weight])	Compute the clustering coefficient for nodes.
average_clustering(G[, nodes, weight, ...])	Compute the average clustering coefficient for the graph G.
square_clustering(G[, nodes])	Compute the squares clustering coefficient for nodes.
generalized_degree(G[, nodes])	Compute the generalized degree for nodes.

3.11.1 triangles

triangles(*G*, *nodes=None*)

Compute the number of triangles.

Finds the number of triangles that include a node as one vertex.

Parameters

G
[graph] A networkx graph

nodes
[container of nodes, optional (default= all nodes in G)] Compute triangles for nodes in this container.

Returns

out
[dictionary] Number of triangles keyed by node label.

Notes

When computing triangles for the entire graph each triangle is counted three times, once at each node. Self loops are ignored.

Examples

```
>>> G = nx.complete_graph(5)
>>> print(nx.triangles(G, 0))
6
>>> print(nx.triangles(G))
{0: 6, 1: 6, 2: 6, 3: 6, 4: 6}
>>> print(list(nx.triangles(G, (0, 1)).values()))
[6, 6]
```

3.11.2 transitivity

transitivity(*G*)

Compute graph transitivity, the fraction of all possible triangles present in G.

Possible triangles are identified by the number of "triads" (two edges with a shared vertex).

The transitivity is

$$T = 3\frac{\#triangles}{\#triads}.$$

Parameters

G
[graph]

Returns

out
[float] Transitivity

Examples

```
>>> G = nx.complete_graph(5)
>>> print(nx.transitivity(G))
1.0
```

3.11.3 clustering

clustering(*G*, *nodes=None*, *weight=None*)

Compute the clustering coefficient for nodes.

For unweighted graphs, the clustering of a node u is the fraction of possible triangles through that node that exist,

$$c_u = \frac{2T(u)}{deg(u)(deg(u)-1)},$$

where $T(u)$ is the number of triangles through node u and $deg(u)$ is the degree of u.

For weighted graphs, there are several ways to define clustering [1]. the one used here is defined as the geometric average of the subgraph edge weights [2],

$$c_u = \frac{1}{deg(u)(deg(u)-1))}\sum_{vw}(\hat{w}_{uv}\hat{w}_{uw}\hat{w}_{vw})^{1/3}.$$

The edge weights $\hat{w}_{uv}$ are normalized by the maximum weight in the network $\hat{w}_{uv} = w_{uv}/\max(w)$.

The value of c_u is assigned to 0 if $deg(u) < 2$.

Additionally, this weighted definition has been generalized to support negative edge weights [3].

For directed graphs, the clustering is similarly defined as the fraction of all possible directed triangles or geometric average of the subgraph edge weights for unweighted and weighted directed graph respectively [4].

$$c_u = \frac{2}{deg^{tot}(u)(deg^{tot}(u) - 1) - 2deg^{\leftrightarrow}(u)} T(u),$$

where $T(u)$ is the number of directed triangles through node u, $deg^{tot}(u)$ is the sum of in degree and out degree of u and $deg^{\leftrightarrow}(u)$ is the reciprocal degree of u.

Parameters

G
[graph]

nodes
[container of nodes, optional (default=all nodes in G)] Compute clustering for nodes in this container.

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

out
[float, or dictionary] Clustering coefficient at specified nodes

Notes

Self loops are ignored.

References

[1], [2], [3], [4]

Examples

```
>>> G = nx.complete_graph(5)
>>> print(nx.clustering(G, 0))
1.0
>>> print(nx.clustering(G))
{0: 1.0, 1: 1.0, 2: 1.0, 3: 1.0, 4: 1.0}
```

3.11.4 average_clustering

average_clustering(*G*, *nodes=None*, *weight=None*, *count_zeros=True*)

Compute the average clustering coefficient for the graph G.

The clustering coefficient for the graph is the average,

$$C = \frac{1}{n}\sum_{v \in G} c_v,$$

where n is the number of nodes in G.

Parameters

G
[graph]

nodes
[container of nodes, optional (default=all nodes in G)] Compute average clustering for nodes in this container.

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.

count_zeros
[bool] If False include only the nodes with nonzero clustering in the average.

Returns

avg
[float] Average clustering

Notes

This is a space saving routine; it might be faster to use the clustering function to get a list and then take the average.

Self loops are ignored.

References

[1], [2]

Examples

```
>>> G = nx.complete_graph(5)
>>> print(nx.average_clustering(G))
1.0
```

3.11.5 square_clustering

square_clustering(*G*, *nodes=None*)

Compute the squares clustering coefficient for nodes.

For each node return the fraction of possible squares that exist at the node [1]

$$C_4(v) = \frac{\sum_{u=1}^{k_v} \sum_{w=u+1}^{k_v} q_v(u,w)}{\sum_{u=1}^{k_v} \sum_{w=u+1}^{k_v} [a_v(u,w) + q_v(u,w)]},$$

where $q_v(u,w)$ are the number of common neighbors of u and w other than v (ie squares), and $a_v(u,w) = (k_u - (1 + q_v(u,w) + \theta_{uv})) + (k_w - (1 + q_v(u,w) + \theta_{uw}))$, where $\theta_{uw} = 1$ if u and 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.

Notes

While $C_3(v)$ (triangle clustering) gives the probability that two neighbors of node v are connected with each other, $C_4(v)$ 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], [2]

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}
```

3.11.6 generalized_degree

generalized_degree(*G*, *nodes=None*)

Compute the generalized degree for nodes.

For each node, the generalized degree shows how many edges of given triangle multiplicity the node is connected to. The triangle multiplicity of an edge is the number of triangles an edge participates in. The generalized degree of node i can be written as a vector $\mathbf{k}_i = (k_i^{(0)}, \ldots, k_i^{(N-2)})$ where $k_i^{(j)}$ is the number of edges attached to node i that participate in j triangles.

Parameters

G
[graph]

nodes
[container of nodes, optional (default=all nodes in G)] Compute the generalized degree for nodes in this container.

Returns

out
[Counter, or dictionary of Counters] Generalized degree of specified nodes. The Counter is keyed by edge triangle multiplicity.

Notes

In a network of N nodes, the highest triangle multiplicty an edge can have is N-2.

The return value does not include a `zero` entry if no edges of a particular triangle multiplicity are present.

The number of triangles node i is attached to can be recovered from the generalized degree $\mathbf{k}_i = (k_i^{(0)}, \ldots, k_i^{(N-2)})$ by $(k_i^{(1)} + 2k_i^{(2)} + \ldots + (N-2)k_i^{(N-2)})/2$.

References

[1]

Examples

```
>>> G = nx.complete_graph(5)
>>> print(nx.generalized_degree(G, 0))
Counter({3: 4})
>>> print(nx.generalized_degree(G))
{0: Counter({3: 4}), 1: Counter({3: 4}), 2: Counter({3: 4}), 3: Counter({3: 4}),␣
↪4: Counter({3: 4})}
```

To recover the number of triangles attached to a node:

```
>>> k1 = nx.generalized_degree(G, 0)
>>> sum([k * v for k, v in k1.items()]) / 2 == nx.triangles(G, 0)
True
```

3.12 Coloring

greedy_color(G[, strategy, interchange])	Color a graph using various strategies of greedy graph coloring.
equitable_color(G, num_colors)	Provides equitable (r + 1)-coloring for nodes of G in O(r * n^2) time if deg(G) <= r.

3.12.1 greedy_color

greedy_color(*G*, *strategy='largest_first'*, *interchange=False*)

Color a graph using various strategies of greedy graph coloring.

Attempts to color a graph using as few colors as possible, where no neighbours of a node can have same color as the node itself. The given strategy determines the order in which nodes are colored.

The strategies are described in [1], and smallest-last is based on [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.

Raises

NetworkXPointlessConcept
If `strategy` is `saturation_largest_first` or `independent_set` and `interchange` is `True`.

References

[1], [2], [3]

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
```

3.12.2 equitable_color

equitable_color(*G*, *num_colors*)

Provides equitable (r + 1)-coloring for nodes of G in O(r * n^2) time if deg(G) <= r. The algorithm is described in [1].

Attempts to color a graph using r 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.

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.

Raises

NetworkXAlgorithmError
If the maximum degree of the graph `G` is greater than `num_colors`.

References

[1]

Examples

```
>>> G = nx.cycle_graph(4)
>>> d = nx.coloring.equitable_color(G, num_colors=3)
>>> nx.algorithms.coloring.equitable_coloring.is_equitable(G, d)
True
```

Some node ordering strategies are provided for use with *greedy_color()*.

strategy_connected_sequential(G, colors[, ...])	Returns an iterable over nodes in G in the order given by a breadth-first or depth-first traversal.
strategy_connected_sequential_dfs(G, colors)	Returns an iterable over nodes in G in the order given by a depth-first traversal.
strategy_connected_sequential_bfs(G, colors)	Returns an iterable over nodes in G in the order given by a breadth-first traversal.
strategy_independent_set(G, colors)	Uses a greedy independent set removal strategy to determine the colors.
strategy_largest_first(G, colors)	Returns a list of the nodes of G in decreasing order by degree.
strategy_random_sequential(G, colors[, seed])	Returns a random permutation of the nodes of G as a list.
strategy_saturation_largest_first(G, colors)	Iterates over all the nodes of G in "saturation order" (also known as "DSATUR").
strategy_smallest_last(G, colors)	Returns a deque of the nodes of G, "smallest" last.

3.12.3 strategy_connected_sequential

strategy_connected_sequential(*G*, *colors*, *traversal='bfs'*)

Returns an iterable over nodes in G in the order given by a breadth-first or depth-first traversal.

traversal must be one of the strings 'dfs' or 'bfs', representing depth-first traversal or breadth-first traversal, respectively.

The generated sequence has the property that for each node except the first, at least one neighbor appeared earlier in the sequence.

G is a NetworkX graph. colors is ignored.

3.12.4 strategy_connected_sequential_dfs

strategy_connected_sequential_dfs(*G*, *colors*)

Returns an iterable over nodes in G in the order given by a depth-first traversal.

The generated sequence has the property that for each node except the first, at least one neighbor appeared earlier in the sequence.

G is a NetworkX graph. colors is ignored.

3.12.5 strategy_connected_sequential_bfs

strategy_connected_sequential_bfs(*G*, *colors*)

Returns an iterable over nodes in `G` in the order given by a breadth-first traversal.

The generated sequence has the property that for each node except the first, at least one neighbor appeared earlier in the sequence.

`G` is a NetworkX graph. `colors` is ignored.

3.12.6 strategy_independent_set

strategy_independent_set(*G*, *colors*)

Uses a greedy independent set removal strategy to determine the colors.

This function updates `colors` **in-place** and return `None`, unlike the other strategy functions in this module.

This algorithm repeatedly finds and removes a maximal independent set, assigning each node in the set an unused color.

`G` is a NetworkX graph.

This strategy is related to *`strategy_smallest_last()`*: in that strategy, an independent set of size one is chosen at each step instead of a maximal independent set.

3.12.7 strategy_largest_first

strategy_largest_first(*G*, *colors*)

Returns a list of the nodes of `G` in decreasing order by degree.

`G` is a NetworkX graph. `colors` is ignored.

3.12.8 strategy_random_sequential

strategy_random_sequential(*G*, *colors*, *seed=None*)

Returns a random permutation of the nodes of `G` as a list.

`G` is a NetworkX graph. `colors` is ignored.

seed
: [integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

3.12.9 strategy_saturation_largest_first

strategy_saturation_largest_first(*G*, *colors*)

Iterates over all the nodes of `G` in "saturation order" (also known as "DSATUR").

`G` is a NetworkX graph. `colors` is a dictionary mapping nodes of `G` to colors, for those nodes that have already been colored.

3.12.10 strategy_smallest_last

strategy_smallest_last(*G*, *colors*)

Returns a deque of the nodes of `G`, "smallest" last.

Specifically, the degrees of each node are tracked in a bucket queue. From this, the node of minimum degree is repeatedly popped from the graph, updating its neighbors' degrees.

`G` is a NetworkX graph. `colors` is ignored.

This implementation of the strategy runs in $O(n + m)$ time (ignoring polylogarithmic factors), where n is the number of nodes and m is the number of edges.

This strategy is related to *strategy_independent_set()*: if we interpret each node removed as an independent set of size one, then this strategy chooses an independent set of size one instead of a maximal independent set.

3.13 Communicability

Communicability.

communicability(G)	Returns communicability between all pairs of nodes in G.
communicability_exp(G)	Returns communicability between all pairs of nodes in G.

3.13.1 communicability

communicability(*G*)

Returns communicability between all pairs of nodes in G.

The communicability between pairs of nodes 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_exp
Communicability between all pairs of nodes in G using spectral decomposition.

communicability_betweenness_centrality
Communicability betweeness centrality for each node in G.

Notes

This algorithm uses a spectral decomposition 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` based on the graph spectrum is [1]

$$C(u,v) = \sum_{j=1}^{n} \phi_j(u)\phi_j(v)e^{\lambda_j},$$

where `phi_{j}(u)` is the `urm{th}` element of the `jrm{th}` orthonormal eigenvector of the adjacency matrix associated with the eigenvalue `lambda_{j}`.

References

[1]

Examples

```
>>> G = nx.Graph([(0, 1), (1, 2), (1, 5), (5, 4), (2, 4), (2, 3), (4, 3), (3, 6)])
>>> c = nx.communicability(G)
```

3.13.2 communicability_exp

communicability_exp(*G*)

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 betweeness 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],

$$C(u,v) = (e^A)_{uv},$$

where `A` is the adjacency matrix of G.

References

[1]

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)
```

3.14 Communities

Functions for computing and measuring community structure.

The functions in this class are not imported into the top-level `networkx` namespace. You can access these functions by importing the *networkx.algorithms.community* module, then accessing the functions as attributes of `community`. For example:

```
>>> from networkx.algorithms import community
>>> G = nx.barbell_graph(5, 1)
>>> communities_generator = community.girvan_newman(G)
>>> top_level_communities = next(communities_generator)
>>> next_level_communities = next(communities_generator)
>>> sorted(map(sorted, next_level_communities))
[[0, 1, 2, 3, 4], [5], [6, 7, 8, 9, 10]]
```

3.14.1 Bipartitions

Functions for computing the Kernighan–Lin bipartition algorithm.

kernighan_lin_bisection(G[, partition, ...])	Partition a graph into two blocks using the Kernighan–Lin algorithm.

kernighan_lin_bisection

kernighan_lin_bisection(*G*, *partition=None*, *max_iter=10*, *weight='weight'*, *seed=None*)

Partition a graph into two blocks using the Kernighan–Lin algorithm.

This algorithm partitions a network into two sets by iteratively swapping pairs of nodes to reduce the edge cut between the two sets. The pairs are chosen according to a modified form of Kernighan-Lin, which moves node individually, alternating between sides to keep the bisection balanced.

Parameters

G
[graph]

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 improvemement 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 *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]

3.14.2 K-Clique

k_clique_communities(G, k[, cliques])	Find k-clique communities in graph using the percolation method.

k_clique_communities

k_clique_communities(*G*, *k*, *cliques=None*)

Find k-clique communities in graph using the percolation method.

A k-clique community is the union of all cliques of size k that can be reached through adjacent (sharing k-1 nodes) k-cliques.

Parameters

G
[NetworkX graph]

k
[int] Size of smallest clique

cliques: list or generator
Precomputed cliques (use networkx.find_cliques(G))

Returns

Yields sets of nodes, one for each k-clique community.

References

[1]

Examples

```
>>> from networkx.algorithms.community import k_clique_communities
>>> G = nx.complete_graph(5)
>>> K5 = nx.convert_node_labels_to_integers(G, first_label=2)
>>> G.add_edges_from(K5.edges())
>>> c = list(k_clique_communities(G, 4))
>>> sorted(list(c[0]))
[0, 1, 2, 3, 4, 5, 6]
>>> list(k_clique_communities(G, 6))
[]
```

3.14.3 Modularity-based communities

Functions for detecting communities based on modularity.

greedy_modularity_communities(G[, weight, ...])	Find communities in G using greedy modularity maximization.
naive_greedy_modularity_communities(G[, ...])	Find communities in G using greedy modularity maximization.

greedy_modularity_communities

greedy_modularity_communities(*G*, *weight=None*, *resolution=1*, *cutoff=1*, *best_n=None*)

Find communities in G using greedy modularity maximization.

This function uses Clauset-Newman-Moore greedy modularity maximization [2] to find the community partition with the largest modularity.

Greedy modularity maximization begins with each node in its own community and repeatedly joins the pair of communities that lead to the largest modularity until no futher increase in modularity is possible (a maximum). Two keyword arguments adjust the stopping condition. `cutoff` is a lower limit on the number of communities so you can stop the process before reaching a maximum (used to save computation time). `best_n` is an upper limit on the number of communities so you can make the process continue until at most n communities remain even if the maximum modularity occurs for more. To obtain exactly n communities, set both `cutoff` and `best_n` to n.

This function maximizes the generalized modularity, where `resolution` is the resolution parameter, often expressed as γ. See *`modularity()`*.

Parameters

G
[NetworkX graph]

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.

resolution
[float, optional (default=1)] If resolution is less than 1, modularity favors larger communities. Greater than 1 favors smaller communities.

cutoff
[int, optional (default=1)] A minimum number of communities below which the merging process stops. The process stops at this number of communities even if modularity is not maximized. The goal is to let the user stop the process early. The process stops before the cutoff if it finds a maximum of modularity.

best_n
[int or None, optional (default=None)] A maximum number of communities above which the merging process will not stop. This forces community merging to continue after modularity starts to decrease until `best_n` communities remain. If `None`, don't force it to continue beyond a maximum.

Returns

communities: list
A list of frozensets of nodes, one for each community. Sorted by length with largest communities first.

Raises

ValueError
[If the `cutoff` or `best_n` value is not in the range] `[1, G.number_of_nodes()]`, or if `best_n` < `cutoff`.

See also:

modularity

References

[1], [2], [3], [4]

Examples

```
>>> from networkx.algorithms.community import greedy_modularity_communities
>>> G = nx.karate_club_graph()
>>> c = greedy_modularity_communities(G)
>>> sorted(c[0])
[8, 14, 15, 18, 20, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33]
```

naive_greedy_modularity_communities

naive_greedy_modularity_communities(*G*, *resolution=1*, *weight=None*)

Find communities in G using greedy modularity maximization.

This implementation is O(n^4), much slower than alternatives, but it is provided as an easy-to-understand reference implementation.

Greedy modularity maximization begins with each node in its own community and joins the pair of communities that most increases modularity until no such pair exists.

This function maximizes the generalized modularity, where `resolution` is the resolution parameter, often expressed as γ. See *modularity()*.

Parameters

G
[NetworkX graph]

resolution
[float (default=1)] If resolution is less than 1, modularity favors larger communities. Greater than 1 favors smaller communities.

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

list
A list of sets of nodes, one for each community. Sorted by length with largest communities first.

See also:

greedy_modularity_communities
modularity

Examples

```
>>> from networkx.algorithms.community import \
... naive_greedy_modularity_communities
>>> G = nx.karate_club_graph()
>>> c = naive_greedy_modularity_communities(G)
>>> sorted(c[0])
[8, 14, 15, 18, 20, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33]
```

3.14.4 Tree partitioning

Lukes Algorithm for exact optimal weighted tree partitioning.

lukes_partitioning(G, max_size[, ...])	Optimal partitioning of a weighted tree using the Lukes algorithm.

lukes_partitioning

lukes_partitioning(*G*, *max_size*, *node_weight=None*, *edge_weight=None*)

Optimal partitioning of a weighted tree using the Lukes algorithm.

This algorithm partitions a connected, acyclic graph featuring integer node weights and float edge weights. The resulting clusters are such that the total weight of the nodes in each cluster does not exceed max_size and that the weight of the edges that are cut by the partition is minimum. The algorithm is based on LUKES[1].

Parameters

G
[graph]

max_size
[int] Maximum weight a partition can have in terms of sum of node_weight for all nodes in the partition

edge_weight
[key] Edge data key to use as weight. If None, the weights are all set to one.

node_weight
[key] Node data key to use as weight. If None, the weights are all set to one. The data must be int.

Returns

partition
[list] A list of sets of nodes representing the clusters of the partition.

Raises

NotATree
If G is not a tree.

TypeError
If any of the values of node_weight is not int.

References

3.14.5 Label propagation

Label propagation community detection algorithms.

asyn_lpa_communities(G[, weight, seed])	Returns communities in G as detected by asynchronous label propagation.
label_propagation_communities(G)	Generates community sets determined by label propagation

asyn_lpa_communities

asyn_lpa_communities(*G*, *weight=None*, *seed=None*)

Returns communities in G as detected by asynchronous label propagation.

The asynchronous label propagation algorithm is described in [1]. The algorithm is probabilistic and the found communities may vary on different executions.

The algorithm proceeds as follows. After initializing each node with a unique label, the algorithm repeatedly sets the label of a node to be the label that appears most frequently among that nodes neighbors. The algorithm halts when each node has the label that appears most frequently among its neighbors. The algorithm is asynchronous because each node is updated without waiting for updates on the remaining nodes.

This generalized version of the algorithm in [1] accepts edge weights.

Parameters

G
[Graph]

weight
[string] The edge attribute representing the weight of an edge. If None, each edge is assumed to have weight one. In this algorithm, the weight of an edge is used in determining the frequency with which a label appears among the neighbors of a node: a higher weight means the label appears more often.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

communities
[iterable] Iterable of communities given as sets of nodes.

Notes

Edge weight attributes must be numerical.

References

[1]

label_propagation_communities

label_propagation_communities(*G*)

Generates community sets determined by label propagation

Finds communities in G using a semi-synchronous label propagation method [1]. This method combines the advantages of both the synchronous and asynchronous models. Not implemented for directed graphs.

Parameters

G
[graph] An undirected NetworkX graph.

Returns

communities
[iterable] A dict_values object that contains a set of nodes for each community.

Raises

NetworkXNotImplemented
If the graph is directed

References

[1]

3.14.6 Louvain Community Detection

Function for detecting communities based on Louvain Community Detection Algorithm

louvain_communities(G[, weight, resolution, ...])	Find the best partition of a graph using the Louvain Community Detection Algorithm.
louvain_partitions(G[, weight, resolution, ...])	Yields partitions for each level of the Louvain Community Detection Algorithm

louvain_communities

louvain_communities(*G*, *weight='weight'*, *resolution=1*, *threshold=1e-07*, *seed=None*)

Find the best partition of a graph using the Louvain Community Detection Algorithm.

Louvain Community Detection Algorithm is a simple method to extract the community structure of a network. This is a heuristic method based on modularity optimization. [1]

The algorithm works in 2 steps. On the first step it assigns every node to be in its own community and then for each node it tries to find the maximum positive modularity gain by moving each node to all of its neighbor communities. If no positive gain is achieved the node remains in its original community.

The modularity gain obtained by moving an isolated node i into a community C can easily be calculated by the following formula (combining [1] [2] and some algebra):

$$\Delta Q = \frac{k_{i,in}}{2m} - \gamma \frac{\Sigma_{tot} \cdot k_i}{2m^2}$$

where m is the size of the graph, $k_{i,in}$ is the sum of the weights of the links from i to nodes in C, k_i is the sum of the weights of the links incident to node i, Σ_{tot} is the sum of the weights of the links incident to nodes in C and γ is the resolution parameter.

For the directed case the modularity gain can be computed using this formula according to [3]

$$\Delta Q = \frac{k_{i,in}}{m} - \gamma \frac{k_i^{out} \cdot \Sigma_{tot}^{in} + k_i^{in} \cdot \Sigma_{tot}^{out}}{m^2}$$

where k_i^{out}, k_i^{in} are the outer and inner weighted degrees of node i and Σ_{tot}^{in}, Σ_{tot}^{out} are the sum of in-going and out-going links incident to nodes in C.

The first phase continues until no individual move can improve the modularity.

The second phase consists in building a new network whose nodes are now the communities found in the first phase. To do so, the weights of the links between the new nodes are given by the sum of the weight of the links between nodes in the corresponding two communities. Once this phase is complete it is possible to reapply the first phase creating bigger communities with increased modularity.

The above two phases are executed until no modularity gain is achieved (or is less than the `threshold`).

Parameters

G
[NetworkX 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.

resolution
[float, optional (default=1)] If resolution is less than 1, the algorithm favors larger communities. Greater than 1 favors smaller communities

threshold
[float, optional (default=0.0000001)] Modularity gain threshold for each level. If the gain of modularity between 2 levels of the algorithm is less than the given threshold then the algorithm stops and returns the resulting communities.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

list
A list of sets (partition of G). Each set represents one community and contains all the nodes that constitute it.

See also:

louvain_partitions

Notes

The order in which the nodes are considered can affect the final output. In the algorithm the ordering happens using a random shuffle.

References

[1], [2], [3]

Examples

```
>>> import networkx as nx
>>> import networkx.algorithms.community as nx_comm
>>> G = nx.petersen_graph()
>>> nx_comm.louvain_communities(G, seed=123)
[{0, 4, 5, 7, 9}, {1, 2, 3, 6, 8}]
```

louvain_partitions

louvain_partitions(*G*, *weight='weight'*, *resolution=1*, *threshold=1e-07*, *seed=None*)

Yields partitions for each level of the Louvain Community Detection Algorithm

Louvain Community Detection Algorithm is a simple method to extract the community structure of a network. This is a heuristic method based on modularity optimization. [1]

The partitions at each level (step of the algorithm) form a dendogram of communities. A dendrogram is a diagram representing a tree and each level represents a partition of the G graph. The top level contains the smallest communities and as you traverse to the bottom of the tree the communities get bigger and the overal modularity increases making the partition better.

Each level is generated by executing the two phases of the Louvain Community Detection Algorithm.

Parameters

G
[NetworkX 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.

resolution
[float, optional (default=1)] If resolution is less than 1, the algorithm favors larger communities. Greater than 1 favors smaller communities

threshold
[float, optional (default=0.0000001)] Modularity gain threshold for each level. If the gain of modularity between 2 levels of the algorithm is less than the given threshold then the algorithm stops and returns the resulting communities.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Yields

list
A list of sets (partition of `G`). Each set represents one community and contains all the nodes that constitute it.

See also:

louvain_communities

References

[1]

3.14.7 Fluid Communities

Asynchronous Fluid Communities algorithm for community detection.

asyn_fluidc(G, k[, max_iter, seed])	Returns communities in `G` as detected by Fluid Communities algorithm.

asyn_fluidc

asyn_fluidc(*G*, *k*, *max_iter=100*, *seed=None*)

Returns communities in `G` as detected by Fluid Communities algorithm.

The asynchronous fluid communities algorithm is described in [1]. The algorithm is based on the simple idea of fluids interacting in an environment, expanding and pushing each other. Its initialization is random, so found communities may vary on different executions.

The algorithm proceeds as follows. First each of the initial k communities is initialized in a random vertex in the graph. Then the algorithm iterates over all vertices in a random order, updating the community of each vertex based on its own community and the communities of its neighbours. This process is performed several times until convergence. At all times, each community has a total density of 1, which is equally distributed among the vertices it contains. If a vertex changes of community, vertex densities of affected communities are adjusted immediately. When a complete iteration over all vertices is done, such that no vertex changes the community it belongs to, the algorithm has converged and returns.

This is the original version of the algorithm described in [1]. Unfortunately, it does not support weighted graphs yet.

Parameters

G
[Graph]

k
[integer] The number of communities to be found.

max_iter
[integer] The number of maximum iterations allowed. By default 100.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

communities
[iterable] Iterable of communities given as sets of nodes.

Notes

k variable is not an optional argument.

References

[1]

3.14.8 Measuring partitions

Functions for measuring the quality of a partition (into communities).

modularity(G, communities[, weight, resolution])	Returns the modularity of the given partition of the graph.
partition_quality(G, partition)	Returns the coverage and performance of a partition of G.

modularity

modularity(*G*, *communities*, *weight='weight'*, *resolution=1*)

Returns the modularity of the given partition of the graph.

Modularity is defined in [1] as

$$Q = \frac{1}{2m} \sum_{ij} \left(A_{ij} - \gamma \frac{k_i k_j}{2m} \right) \delta(c_i, c_j)$$

where m is the number of edges, A is the adjacency matrix of `G`, k_i is the degree of i, γ 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

$$Q = \sum_{c=1}^{n} \left[\frac{L_c}{m} - \gamma \left(\frac{k_c}{2m} \right)^2 \right]$$

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 γ 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 $k_c^{in} k_c^{out}$.

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 paritition.

Raises

NotAPartition
If `communities` is not a partition of the nodes of `G`.

References

[1], [2], [3], [4]

Examples

```
>>> import networkx.algorithms.community as nx_comm
>>> G = nx.barbell_graph(3, 0)
>>> nx_comm.modularity(G, [{0, 1, 2}, {3, 4, 5}])
0.35714285714285715
>>> nx_comm.modularity(G, nx_comm.label_propagation_communities(G))
0.35714285714285715
```

partition_quality

partition_quality(*G*, *partition*)

Returns the coverage and performance of a partition of G.

The *coverage* of a partition is the ratio of the number of intra-community edges to the total number of edges in the graph.

The *performance* of a partition is the number of intra-community edges plus inter-community non-edges divided by the total number of potential edges.

This algorithm has complexity $O(C^2 + L)$ where C is the number of communities and L is the number of links.

Parameters

G
[NetworkX graph]

partition
[sequence] Partition of the nodes of `G`, represented as a sequence of sets of nodes (blocks). Each block of the partition represents a community.

Returns

(float, float)
The (coverage, performance) tuple of the partition, as defined above.

Raises

NetworkXError
If `partition` is not a valid partition of the nodes of `G`.

Notes

If `G` is a multigraph;

- for coverage, the multiplicity of edges is counted
- for performance, the result is -1 (total number of possible edges is not defined)

References

[1]

3.14.9 Partitions via centrality measures

Functions for computing communities based on centrality notions.

girvan_newman(G[, most_valuable_edge])	Finds communities in a graph using the Girvan–Newman method.

girvan_newman

girvan_newman(*G*, *most_valuable_edge=None*)

Finds communities in a graph using the Girvan–Newman method.

Parameters

G
[NetworkX graph]

most_valuable_edge
[function] Function that takes a graph as input and outputs an edge. The edge returned by this function will be recomputed and removed at each iteration of the algorithm.

If not specified, the edge with the highest `networkx.edge_betweenness_centrality()` will be used.

Returns

iterator
Iterator over tuples of sets of nodes in `G`. Each set of node is a community, each tuple is a sequence of communities at a particular level of the algorithm.

Notes

The Girvan–Newman algorithm detects communities by progressively removing edges from the original graph. The algorithm removes the "most valuable" edge, traditionally the edge with the highest betweenness centrality, at each step. As the graph breaks down into pieces, the tightly knit community structure is exposed and the result can be depicted as a dendrogram.

Examples

To get the first pair of communities:

```
>>> G = nx.path_graph(10)
>>> comp = girvan_newman(G)
>>> tuple(sorted(c) for c in next(comp))
([0, 1, 2, 3, 4], [5, 6, 7, 8, 9])
```

To get only the first k tuples of communities, use `itertools.islice()`:

```
>>> import itertools
>>> G = nx.path_graph(8)
>>> k = 2
>>> comp = girvan_newman(G)
>>> for communities in itertools.islice(comp, k):
...     print(tuple(sorted(c) for c in communities))
...
([0, 1, 2, 3], [4, 5, 6, 7])
([0, 1], [2, 3], [4, 5, 6, 7])
```

To stop getting tuples of communities once the number of communities is greater than k, use `itertools.takewhile()`:

```
>>> import itertools
>>> G = nx.path_graph(8)
>>> k = 4
>>> comp = girvan_newman(G)
>>> limited = itertools.takewhile(lambda c: len(c) <= k, comp)
>>> for communities in limited:
...     print(tuple(sorted(c) for c in communities))
...
([0, 1, 2, 3], [4, 5, 6, 7])
([0, 1], [2, 3], [4, 5, 6, 7])
([0, 1], [2, 3], [4, 5], [6, 7])
```

To just choose an edge to remove based on the weight:

```
>>> from operator import itemgetter
>>> G = nx.path_graph(10)
>>> edges = G.edges()
>>> nx.set_edge_attributes(G, {(u, v): v for u, v in edges}, "weight")
>>> def heaviest(G):
...     u, v, w = max(G.edges(data="weight"), key=itemgetter(2))
```

(continues on next page)

(continued from previous page)

```
...     return (u, v)
...
>>> comp = girvan_newman(G, most_valuable_edge=heaviest)
>>> tuple(sorted(c) for c in next(comp))
([0, 1, 2, 3, 4, 5, 6, 7, 8], [9])
```

To utilize edge weights when choosing an edge with, for example, the highest betweenness centrality:

```
>>> from networkx import edge_betweenness_centrality as betweenness
>>> def most_central_edge(G):
...     centrality = betweenness(G, weight="weight")
...     return max(centrality, key=centrality.get)
...
>>> G = nx.path_graph(10)
>>> comp = girvan_newman(G, most_valuable_edge=most_central_edge)
>>> tuple(sorted(c) for c in next(comp))
([0, 1, 2, 3, 4], [5, 6, 7, 8, 9])
```

To specify a different ranking algorithm for edges, use the `most_valuable_edge` keyword argument:

```
>>> from networkx import edge_betweenness_centrality
>>> from random import random
>>> def most_central_edge(G):
...     centrality = edge_betweenness_centrality(G)
...     max_cent = max(centrality.values())
...     # Scale the centrality values so they are between 0 and 1,
...     # and add some random noise.
...     centrality = {e: c / max_cent for e, c in centrality.items()}
...     # Add some random noise.
...     centrality = {e: c + random() for e, c in centrality.items()}
...     return max(centrality, key=centrality.get)
...
>>> G = nx.path_graph(10)
>>> comp = girvan_newman(G, most_valuable_edge=most_central_edge)
```

3.14.10 Validating partitions

Helper functions for community-finding algorithms.

is_partition(G, communities)	Returns *True* if `communities` is a partition of the nodes of `G`.

is_partition

is_partition(*G*, *communities*)

Returns *True* if `communities` is a partition of the nodes of `G`.

A partition of a universe set is a family of pairwise disjoint sets whose union is the entire universe set.

Parameters

G

[NetworkX graph.]

communities
[list or iterable of sets of nodes] If not a list, the iterable is converted internally to a list. If it is an iterator it is exhausted.

3.15 Components

3.15.1 Connectivity

is_connected(G)	Returns True if the graph is connected, False otherwise.
number_connected_components(G)	Returns the number of connected components.
connected_components(G)	Generate connected components.
node_connected_component(G, n)	Returns the set of nodes in the component of graph containing node n.

is_connected

is_connected(*G*)

Returns True if the graph is connected, False otherwise.

Parameters

G
[NetworkX Graph] An undirected graph.

Returns

connected
[bool] True if the graph is connected, false otherwise.

Raises

NetworkXNotImplemented
If G is directed.

See also:

is_strongly_connected
is_weakly_connected
is_semiconnected
is_biconnected
connected_components

Notes

For undirected graphs only.

Examples

```
>>> G = nx.path_graph(4)
>>> print(nx.is_connected(G))
True
```

number_connected_components

number_connected_components(*G*)

Returns the number of connected components.

Parameters

G
[NetworkX graph] An undirected graph.

Returns

n
[integer] Number of connected components

See also:

connected_components
number_weakly_connected_components
number_strongly_connected_components

Notes

For undirected graphs only.

Examples

```
>>> G = nx.Graph([(0, 1), (1, 2), (5, 6), (3, 4)])
>>> nx.number_connected_components(G)
3
```

connected_components

connected_components(*G*)

Generate connected components.

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.

See also:

strongly_connected_components
weakly_connected_components

Notes

For undirected graphs only.

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)]
```

node_connected_component

node_connected_component(*G*, *n*)

Returns the set of nodes in the component of graph containing node n.

Parameters

G
[NetworkX Graph] An undirected graph.

n
[node label] A node in G

Returns

comp
[set] A set of nodes in the component of G containing node n.

Raises

NetworkXNotImplemented
If G is directed.

See also:

connected_components

Notes

For undirected graphs only.

Examples

```
>>> G = nx.Graph([(0, 1), (1, 2), (5, 6), (3, 4)])
>>> nx.node_connected_component(G, 0)  # nodes of component that contains node 0
{0, 1, 2}
```

3.15.2 Strong connectivity

is_strongly_connected(G)	Test directed graph for strong connectivity.
number_strongly_connected_components(G)	Returns number of strongly connected components in graph.
strongly_connected_components(G)	Generate nodes in strongly connected components of graph.
strongly_connected_components_recursive(G)	Generate nodes in strongly connected components of graph.
kosaraju_strongly_connected_components(G[, ...])	Generate nodes in strongly connected components of graph.
condensation(G[, scc])	Returns the condensation of G.

is_strongly_connected

is_strongly_connected(*G*)

Test directed graph for strong connectivity.

A directed graph is strongly connected if and only if every vertex in the graph is reachable from every other vertex.

Parameters

G
[NetworkX Graph] A directed graph.

Returns

connected
[bool] True if the graph is strongly connected, False otherwise.

Raises

NetworkXNotImplemented
If G is undirected.

See also:

is_weakly_connected
is_semiconnected
is_connected
is_biconnected
strongly_connected_components

Notes

For directed graphs only.

Examples

```
>>> G = nx.DiGraph([(0, 1), (1, 2), (2, 3), (3, 0), (2, 4), (4, 2)])
>>> nx.is_strongly_connected(G)
True
>>> G.remove_edge(2, 3)
>>> nx.is_strongly_connected(G)
False
```

number_strongly_connected_components

number_strongly_connected_components(*G*)

Returns number of strongly connected components in graph.

Parameters

G
[NetworkX graph] A directed graph.

Returns

n
[integer] Number of strongly connected components

Raises

NetworkXNotImplemented
If G is undirected.

See also:

strongly_connected_components
number_connected_components
number_weakly_connected_components

Notes

For directed graphs only.

Examples

```
>>> G = nx.DiGraph([(0, 1), (1, 2), (2, 0), (2, 3), (4, 5), (3, 4), (5, 6), (6,␣
↪3), (6, 7)])
>>> nx.number_strongly_connected_components(G)
3
```

strongly_connected_components

strongly_connected_components(*G*)

Generate nodes in strongly connected components of graph.

Parameters

G
[NetworkX Graph] A directed graph.

Returns

comp
[generator of sets] A generator of sets of nodes, one for each strongly connected component of G.

Raises

NetworkXNotImplemented
If G is undirected.

See also:

connected_components
weakly_connected_components
kosaraju_strongly_connected_components

Notes

Uses Tarjan's algorithm[R827335e01166-1]_ with Nuutila's modifications[R827335e01166-2]_. Nonrecursive version of algorithm.

References

[1], [2]

Examples

Generate a sorted list of strongly connected components, largest first.

```
>>> G = nx.cycle_graph(4, create_using=nx.DiGraph())
>>> nx.add_cycle(G, [10, 11, 12])
>>> [
...     len(c)
...     for c in sorted(nx.strongly_connected_components(G), key=len,␣
↪reverse=True)
... ]
[4, 3]
```

If you only want the largest component, it's more efficient to use max instead of sort.

```
>>> largest = max(nx.strongly_connected_components(G), key=len)
```

strongly_connected_components_recursive

strongly_connected_components_recursive(*G*)

Generate nodes in strongly connected components of graph.

Recursive version of algorithm.

Parameters

G
[NetworkX Graph] A directed graph.

Returns

comp
[generator of sets] A generator of sets of nodes, one for each strongly connected component of G.

Raises

NetworkXNotImplemented
If G is undirected.

See also:

connected_components

Notes

Uses Tarjan's algorithm[Re7cb971df765-1]_ with Nuutila's modifications[Re7cb971df765-2]_.

References

[1], [2]

Examples

Generate a sorted list of strongly connected components, largest first.

```
>>> G = nx.cycle_graph(4, create_using=nx.DiGraph())
>>> nx.add_cycle(G, [10, 11, 12])
>>> [
...     len(c)
...     for c in sorted(
...         nx.strongly_connected_components_recursive(G), key=len, reverse=True
...     )
... ]
[4, 3]
```

If you only want the largest component, it's more efficient to use max instead of sort.

```
>>> largest = max(nx.strongly_connected_components_recursive(G), key=len)
```

To create the induced subgraph of the components use: >>> S = [G.subgraph(c).copy() for c in nx.weakly_connected_components(G)]

kosaraju_strongly_connected_components

kosaraju_strongly_connected_components(*G*, *source=None*)

Generate nodes in strongly connected components of graph.

Parameters

G
[NetworkX Graph] A directed graph.

Returns

comp
[generator of sets] A generator of sets of nodes, one for each strongly connected component of G.

Raises

NetworkXNotImplemented
If G is undirected.

See also:

strongly_connected_components

Notes

Uses Kosaraju's algorithm.

Examples

Generate a sorted list of strongly connected components, largest first.

```
>>> G = nx.cycle_graph(4, create_using=nx.DiGraph())
>>> nx.add_cycle(G, [10, 11, 12])
>>> [
...     len(c)
...     for c in sorted(
...         nx.kosaraju_strongly_connected_components(G), key=len, reverse=True
...     )
... ]
[4, 3]
```

If you only want the largest component, it's more efficient to use max instead of sort.

```
>>> largest = max(nx.kosaraju_strongly_connected_components(G), key=len)
```

condensation

condensation(*G*, *scc=None*)

Returns the condensation of G.

The condensation of G is the graph with each of the strongly connected components contracted into a single node.

Parameters

G
[NetworkX DiGraph] A directed graph.

scc: list or generator (optional, default=None)
Strongly connected components. If provided, the elements in `scc` must partition the nodes in `G`. If not provided, it will be calculated as scc=nx.strongly_connected_components(G).

Returns

C
[NetworkX DiGraph] The condensation graph C of G. The node labels are integers corresponding to the index of the component in the list of strongly connected components of G. C has a graph attribute named 'mapping' with a dictionary mapping the original nodes to the nodes in C to which they belong. Each node in C also has a node attribute 'members' with the set of original nodes in G that form the SCC that the node in C represents.

Raises

NetworkXNotImplemented
If G is undirected.

Notes

After contracting all strongly connected components to a single node, the resulting graph is a directed acyclic graph.

Examples

Contracting two sets of strongly connected nodes into two distinct SCC using the barbell graph.

```
>>> G = nx.barbell_graph(4, 0)
>>> G.remove_edge(3, 4)
>>> G = nx.DiGraph(G)
>>> H = nx.condensation(G)
>>> H.nodes.data()
NodeDataView({0: {'members': {0, 1, 2, 3}}, 1: {'members': {4, 5, 6, 7}}})
>>> H.graph['mapping']
{0: 0, 1: 0, 2: 0, 3: 0, 4: 1, 5: 1, 6: 1, 7: 1}
```

Contracting a complete graph into one single SCC.

```
>>> G = nx.complete_graph(7, create_using=nx.DiGraph)
>>> H = nx.condensation(G)
>>> H.nodes
NodeView((0,))
>>> H.nodes.data()
NodeDataView({0: {'members': {0, 1, 2, 3, 4, 5, 6}}})
```

3.15.3 Weak connectivity

is_weakly_connected(G)	Test directed graph for weak connectivity.
number_weakly_connected_components(G)	Returns the number of weakly connected components in G.
weakly_connected_components(G)	Generate weakly connected components of G.

is_weakly_connected

is_weakly_connected(*G*)

Test directed graph for weak connectivity.

A directed graph is weakly connected if and only if the graph is connected when the direction of the edge between nodes is ignored.

Note that if a graph is strongly connected (i.e. the graph is connected even when we account for directionality), it is by definition weakly connected as well.

Parameters

G
[NetworkX Graph] A directed graph.

Returns

connected
[bool] True if the graph is weakly connected, False otherwise.

Raises

NetworkXNotImplemented
If G is undirected.

See also:

is_strongly_connected
is_semiconnected
is_connected
is_biconnected
weakly_connected_components

Notes

For directed graphs only.

Examples

```
>>> G = nx.DiGraph([(0, 1), (2, 1)])
>>> G.add_node(3)
>>> nx.is_weakly_connected(G)  # node 3 is not connected to the graph
False
>>> G.add_edge(2, 3)
>>> nx.is_weakly_connected(G)
True
```

number_weakly_connected_components

number_weakly_connected_components(*G*)

Returns the number of weakly connected components in G.

Parameters

G
[NetworkX graph] A directed graph.

Returns

n
[integer] Number of weakly connected components

Raises

NetworkXNotImplemented
If G is undirected.

See also:

weakly_connected_components
number_connected_components
number_strongly_connected_components

Notes

For directed graphs only.

Examples

```
>>> G = nx.DiGraph([(0, 1), (2, 1), (3, 4)])
>>> nx.number_weakly_connected_components(G)
2
```

weakly_connected_components

weakly_connected_components(*G*)

Generate weakly connected components of G.

Parameters

G
[NetworkX graph] A directed graph

Returns

comp
[generator of sets] A generator of sets of nodes, one for each weakly connected component of G.

Raises

NetworkXNotImplemented
If G is undirected.

See also:

connected_components
strongly_connected_components

Notes

For directed graphs only.

Examples

Generate a sorted list of weakly connected components, largest first.

```
>>> G = nx.path_graph(4, create_using=nx.DiGraph())
>>> nx.add_path(G, [10, 11, 12])
>>> [
...     len(c)
...     for c in sorted(nx.weakly_connected_components(G), key=len, reverse=True)
... ]
[4, 3]
```

If you only want the largest component, it's more efficient to use max instead of sort:

```
>>> largest_cc = max(nx.weakly_connected_components(G), key=len)
```

3.15.4 Attracting components

is_attracting_component(G)	Returns True if G consists of a single attracting component.
number_attracting_components(G)	Returns the number of attracting components in G.
attracting_components(G)	Generates the attracting components in G.

is_attracting_component

is_attracting_component(*G*)

Returns True if G consists of a single attracting component.

Parameters

G
[DiGraph, MultiDiGraph] The graph to be analyzed.

Returns

attracting
[bool] True if G has a single attracting component. Otherwise, False.

Raises

NetworkXNotImplemented
If the input graph is undirected.

See also:

attracting_components
number_attracting_components

number_attracting_components

number_attracting_components(*G*)

Returns the number of attracting components in G.

Parameters

G
[DiGraph, MultiDiGraph] The graph to be analyzed.

Returns

n
[int] The number of attracting components in G.

Raises

NetworkXNotImplemented
If the input graph is undirected.

See also:

attracting_components
is_attracting_component

attracting_components

attracting_components(*G*)

Generates the attracting components in `G`.

An attracting component in a directed graph `G` is a strongly connected component with the property that a random walker on the graph will never leave the component, once it enters the component.

The nodes in attracting components can also be thought of as recurrent nodes. If a random walker enters the attractor containing the node, then the node will be visited infinitely often.

To obtain induced subgraphs on each component use: `(G.subgraph(c).copy() for c in attracting_components(G))`

Parameters

G
[DiGraph, MultiDiGraph] The graph to be analyzed.

Returns

attractors
[generator of sets] A generator of sets of nodes, one for each attracting component of G.

Raises

NetworkXNotImplemented
If the input graph is undirected.

See also:

number_attracting_components
is_attracting_component

3.15.5 Biconnected components

is_biconnected(G)	Returns True if the graph is biconnected, False otherwise.
biconnected_components(G)	Returns a generator of sets of nodes, one set for each biconnected component of the graph
biconnected_component_edges(G)	Returns a generator of lists of edges, one list for each biconnected component of the input graph.
articulation_points(G)	Yield the articulation points, or cut vertices, of a graph.

is_biconnected

is_biconnected(*G*)

Returns True if the graph is biconnected, False otherwise.

A graph is biconnected if, and only if, it cannot be disconnected by removing only one node (and all edges incident on that node). If removing a node increases the number of disconnected components in the graph, that node is called an articulation point, or cut vertex. A biconnected graph has no articulation points.

Parameters

G
[NetworkX Graph] An undirected graph.

Returns

biconnected
[bool] True if the graph is biconnected, False otherwise.

Raises

NetworkXNotImplemented
If the input graph is not undirected.

See also:

biconnected_components
articulation_points
biconnected_component_edges
is_strongly_connected
is_weakly_connected
is_connected
is_semiconnected

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]

Examples

```
>>> G = nx.path_graph(4)
>>> print(nx.is_biconnected(G))
False
>>> G.add_edge(0, 3)
>>> print(nx.is_biconnected(G))
True
```

biconnected_components

biconnected_components(*G*)

Returns a generator of sets of nodes, one set for each biconnected 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.

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]

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))`

biconnected_component_edges

biconnected_component_edges(G)

Returns a generator of lists of edges, one list for each biconnected component of the input 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. However, each edge belongs to one, and only one, biconnected component.

Notice that by convention a dyad is considered a biconnected component.

Parameters

G
[NetworkX Graph] An undirected graph.

Returns

edges
[generator of lists] Generator of lists of edges, one list for each bicomponent.

Raises

NetworkXNotImplemented
If the input graph is not undirected.

See also:

is_biconnected
biconnected_components
articulation_points

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]

Examples

```
>>> G = nx.barbell_graph(4, 2)
>>> print(nx.is_biconnected(G))
False
>>> bicomponents_edges = list(nx.biconnected_component_edges(G))
>>> len(bicomponents_edges)
5
>>> G.add_edge(2, 8)
>>> print(nx.is_biconnected(G))
True
>>> bicomponents_edges = list(nx.biconnected_component_edges(G))
>>> len(bicomponents_edges)
1
```

articulation_points

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.

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]

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
```

3.15.6 Semiconnectedness

is_semiconnected(G[, topo_order])	Returns True if the graph is semiconnected, False otherwise.

is_semiconnected

is_semiconnected(*G*, *topo_order=None*)

Returns True if the graph is semiconnected, False otherwise.

A graph is semiconnected if, and only if, for any pair of nodes, either one is reachable from the other, or they are mutually reachable.

Parameters

G
[NetworkX graph] A directed graph.

topo_order: list or tuple, optional
A topological order for G (if None, the function will compute one)

Returns

semiconnected
[bool] True if the graph is semiconnected, False otherwise.

Raises

NetworkXNotImplemented
If the input graph is undirected.

NetworkXPointlessConcept
If the graph is empty.

See also:

is_strongly_connected
is_weakly_connected
is_connected
is_biconnected

Examples

```
>>> G = nx.path_graph(4, create_using=nx.DiGraph())
>>> print(nx.is_semiconnected(G))
True
>>> G = nx.DiGraph([(1, 2), (3, 2)])
>>> print(nx.is_semiconnected(G))
False
```

3.16 Connectivity

Connectivity and cut algorithms

3.16.1 Edge-augmentation

Algorithms for finding k-edge-augmentations

A k-edge-augmentation is a set of edges, that once added to a graph, ensures that the graph is k-edge-connected; i.e. the graph cannot be disconnected unless k or more edges are removed. Typically, the goal is to find the augmentation with minimum weight. In general, it is not guaranteed that a k-edge-augmentation exists.

See Also

`edge_kcomponents` : algorithms for finding k-edge-connected components `connectivity` : algorithms for determening edge connectivity.

k_edge_augmentation(G, k[, avail, weight, ...])	Finds set of edges to k-edge-connect G.
is_k_edge_connected(G, k)	Tests to see if a graph is k-edge-connected.
is_locally_k_edge_connected(G, s, t, k)	Tests to see if an edge in a graph is locally k-edge-connected.

k_edge_augmentation

k_edge_augmentation(*G*, *k*, *avail=None*, *weight=None*, *partial=False*)

Finds set of edges to k-edge-connect G.

Adding edges from the augmentation to G make it impossible to disconnect G unless k or more edges are removed. This function uses the most efficient function available (depending on the value of k and if the problem is weighted or unweighted) to search for a minimum weight subset of available edges that k-edge-connects G. In general, finding a k-edge-augmentation is NP-hard, so solutions are not guaranteed to be minimal. Furthermore, a k-edge-augmentation may not exist.

Parameters

G
[NetworkX graph] An undirected graph.

k
[integer] Desired edge connectivity

avail
[dict or a set of 2 or 3 tuples] The available edges that can be used in the augmentation.

If unspecified, then all edges in the complement of G are available. Otherwise, each item is an available edge (with an optional weight).

In the unweighted case, each item is an edge `(u, v)`.

In the weighted case, each item is a 3-tuple `(u, v, d)` or a dict with items `(u, v): d`. The third item, `d`, can be a dictionary or a real number. If `d` is a dictionary `d[weight]` correspondings to the weight.

weight
[string] key to use to find weights if `avail` is a set of 3-tuples where the third item in each tuple is a dictionary.

partial
[boolean] 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 `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)]
```

is_k_edge_connected

is_k_edge_connected(*G*, *k*)

Tests to see if a graph is k-edge-connected.

Is it impossible to disconnect the graph by removing fewer than k edges? If so, then G is k-edge-connected.

Parameters

G
[NetworkX graph] An undirected graph.

k
[integer] edge connectivity to test for

Returns

boolean
True if G is k-edge-connected.

See also:

is_locally_k_edge_connected()

Examples

```
>>> G = nx.barbell_graph(10, 0)
>>> nx.is_k_edge_connected(G, k=1)
True
>>> nx.is_k_edge_connected(G, k=2)
False
```

is_locally_k_edge_connected

is_locally_k_edge_connected(*G*, *s*, *t*, *k*)

Tests to see if an edge in a graph is locally k-edge-connected.

Is it impossible to disconnect s and t by removing fewer than k edges? If so, then s and t are locally k-edge-connected in G.

Parameters

G
[NetworkX graph] An undirected graph.

s
[node] Source node

t
[node] Target node

k
[integer] local edge connectivity for nodes s and t

Returns

boolean
True if s and t are locally k-edge-connected in G.

See also:

is_k_edge_connected()

Examples

```
>>> from networkx.algorithms.connectivity import is_locally_k_edge_connected
>>> G = nx.barbell_graph(10, 0)
>>> is_locally_k_edge_connected(G, 5, 15, k=1)
True
>>> is_locally_k_edge_connected(G, 5, 15, k=2)
False
>>> is_locally_k_edge_connected(G, 1, 5, k=2)
True
```

3.16.2 K-edge-components

Algorithms for finding k-edge-connected components and subgraphs.

A k-edge-connected component (k-edge-cc) is a maximal set of nodes in G, such that all pairs of node have an edge-connectivity of at least k.

A k-edge-connected subgraph (k-edge-subgraph) is a maximal set of nodes in G, such that the subgraph of G defined by the nodes has an edge-connectivity at least k.

k_edge_components(G, k)	Generates nodes in each maximal k-edge-connected component in G.
k_edge_subgraphs(G, k)	Generates nodes in each maximal k-edge-connected subgraph in G.
bridge_components(G)	Finds all bridge-connected components G.
EdgeComponentAuxGraph()	A simple algorithm to find all k-edge-connected components in a graph.

k_edge_components

k_edge_components(*G*, *k*)

Generates nodes in each maximal k-edge-connected component in G.

Parameters

G
[NetworkX graph]

k
[Integer] Desired edge connectivity

Returns

k_edge_components
[a generator of k-edge-ccs. Each set of returned nodes] will have k-edge-connectivity in the graph G.

Raises

NetworkXNotImplemented
If the input graph is a multigraph.

ValueError:
If k is less than 1

See also:

local_edge_connectivity()
k_edge_subgraphs()
similar to this function, but the subgraph defined by the nodes must also have k-edge-connectivity.

k_components()
similar to this function, but uses node-connectivity instead of edge-connectivity

Notes

Attempts to use the most efficient implementation available based on k. If k=1, this is simply connected components for directed graphs and connected components for undirected graphs. If k=2 on an efficient bridge connected component algorithm from _[1] is run based on the chain decomposition. Otherwise, the algorithm from _[2] is used.

References

[1], [2]

Examples

```
>>> import itertools as it
>>> from networkx.utils import pairwise
>>> paths = [
...     (1, 2, 4, 3, 1, 4),
...     (5, 6, 7, 8, 5, 7, 8, 6),
... ]
>>> G = nx.Graph()
>>> G.add_nodes_from(it.chain(*paths))
>>> G.add_edges_from(it.chain(*[pairwise(path) for path in paths]))
>>> # note this returns {1, 4} unlike k_edge_subgraphs
>>> sorted(map(sorted, nx.k_edge_components(G, k=3)))
[[1, 4], [2], [3], [5, 6, 7, 8]]
```

k_edge_subgraphs

k_edge_subgraphs(*G*, *k*)

Generates nodes in each maximal k-edge-connected subgraph in G.

Parameters

G
[NetworkX graph]

k
[Integer] Desired edge connectivity

Returns

k_edge_subgraphs
[a generator of k-edge-subgraphs] Each k-edge-subgraph is a maximal set of nodes that defines a subgraph of G that is k-edge-connected.

Raises

NetworkXNotImplemented
If the input graph is a multigraph.

ValueError:
If k is less than 1

See also:

edge_connectivity()
k_edge_components()
similar to this function, but nodes only need to have k-edge-connctivity within the graph G and the subgraphs might not be k-edge-connected.

Notes

Attempts to use the most efficient implementation available based on k. If k=1, or k=2 and the graph is undirected, then this simply calls *k_edge_components*. Otherwise the algorithm from _[1] is used.

References

[1]

Examples

```
>>> import itertools as it
>>> from networkx.utils import pairwise
>>> paths = [
...     (1, 2, 4, 3, 1, 4),
...     (5, 6, 7, 8, 5, 7, 8, 6),
... ]
>>> G = nx.Graph()
>>> G.add_nodes_from(it.chain(*paths))
>>> G.add_edges_from(it.chain(*[pairwise(path) for path in paths]))
>>> # note this does not return {1, 4} unlike k_edge_components
>>> sorted(map(sorted, nx.k_edge_subgraphs(G, k=3)))
[[1], [2], [3], [4], [5, 6, 7, 8]]
```

bridge_components

bridge_components(*G*)

Finds all bridge-connected components G.

Parameters

G
[NetworkX undirected graph]

Returns

bridge_components
[a generator of 2-edge-connected components]

Raises

NetworkXNotImplemented
If the input graph is directed or a multigraph.

See also:

k_edge_subgraphs()
this function is a special case for an undirected graph where k=2.

biconnected_components()
similar to this function, but is defined using 2-node-connectivity instead of 2-edge-connectivity.

Notes

Bridge-connected components are also known as 2-edge-connected components.

Examples

```
>>> # The barbell graph with parameter zero has a single bridge
>>> G = nx.barbell_graph(5, 0)
>>> from networkx.algorithms.connectivity.edge_kcomponents import bridge_
↪components
>>> sorted(map(sorted, bridge_components(G)))
[[0, 1, 2, 3, 4], [5, 6, 7, 8, 9]]
```

networkx.algorithms.connectivity.edge_kcomponents.EdgeComponentAuxGraph

class EdgeComponentAuxGraph

A simple algorithm to find all k-edge-connected components in a graph.

Constructing the AuxillaryGraph (which may take some time) allows for the k-edge-ccs to be found in linear time for arbitrary k.

Notes

This implementation is based on [1]. The idea is to construct an auxiliary graph from which the k-edge-ccs can be extracted in linear time. The auxiliary graph is constructed in $O(|V| \cdot F)$ operations, where F is the complexity of max flow. Querying the components takes an additional $O(|V|)$ operations. This algorithm can be slow for large graphs, but it handles an arbitrary k and works for both directed and undirected inputs.

The undirected case for k=1 is exactly connected components. The undirected case for k=2 is exactly bridge connected components. The directed case for k=1 is exactly strongly connected components.

References

[1]

Examples

```
>>> import itertools as it
>>> from networkx.utils import pairwise
>>> from networkx.algorithms.connectivity import EdgeComponentAuxGraph
>>> # Build an interesting graph with multiple levels of k-edge-ccs
>>> paths = [
...     (1, 2, 3, 4, 1, 3, 4, 2),  # a 3-edge-cc (a 4 clique)
...     (5, 6, 7, 5),  # a 2-edge-cc (a 3 clique)
...     (1, 5),  # combine first two ccs into a 1-edge-cc
...     (0,),  # add an additional disconnected 1-edge-cc
... ]
>>> G = nx.Graph()
>>> G.add_nodes_from(it.chain(*paths))
>>> G.add_edges_from(it.chain(*[pairwise(path) for path in paths]))
>>> # Constructing the AuxGraph takes about O(n ** 4)
>>> aux_graph = EdgeComponentAuxGraph.construct(G)
>>> # Once constructed, querying takes O(n)
>>> sorted(map(sorted, aux_graph.k_edge_components(k=1)))
[[0], [1, 2, 3, 4, 5, 6, 7]]
>>> sorted(map(sorted, aux_graph.k_edge_components(k=2)))
[[0], [1, 2, 3, 4], [5, 6, 7]]
>>> sorted(map(sorted, aux_graph.k_edge_components(k=3)))
[[0], [1, 2, 3, 4], [5], [6], [7]]
>>> sorted(map(sorted, aux_graph.k_edge_components(k=4)))
[[0], [1], [2], [3], [4], [5], [6], [7]]
```

The auxiliary graph is primarilly used for k-edge-ccs but it can also speed up the queries of k-edge-subgraphs by refining the search space.

```
>>> import itertools as it
>>> from networkx.utils import pairwise
>>> from networkx.algorithms.connectivity import EdgeComponentAuxGraph
>>> paths = [
...     (1, 2, 4, 3, 1, 4),
... ]
>>> G = nx.Graph()
>>> G.add_nodes_from(it.chain(*paths))
>>> G.add_edges_from(it.chain(*[pairwise(path) for path in paths]))
>>> aux_graph = EdgeComponentAuxGraph.construct(G)
>>> sorted(map(sorted, aux_graph.k_edge_subgraphs(k=3)))
```

(continues on next page)

(continued from previous page)

```
[[1], [2], [3], [4]]
>>> sorted(map(sorted, aux_graph.k_edge_components(k=3)))
[[1, 4], [2], [3]]
```

__init__(**args*, ***kwargs*)

Methods

construct(G)	Builds an auxiliary graph encoding edge-connectivity between nodes.
k_edge_components(k)	Queries the auxiliary graph for k-edge-connected components.
k_edge_subgraphs(k)	Queries the auxiliary graph for k-edge-connected subgraphs.

EdgeComponentAuxGraph.construct

classmethod EdgeComponentAuxGraph.**construct**(*G*)

Builds an auxiliary graph encoding edge-connectivity between nodes.

Parameters

G
[NetworkX graph]

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.

EdgeComponentAuxGraph.k_edge_components

EdgeComponentAuxGraph.**k_edge_components**(*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.

EdgeComponentAuxGraph.k_edge_subgraphs

EdgeComponentAuxGraph.**k_edge_subgraphs**(*k*)

Queries the auxiliary graph for k-edge-connected subgraphs.

Parameters

k
[Integer] Desired edge connectivity

Returns

k_edge_subgraphs
[a generator of k-edge-subgraphs]

Notes

Refines the k-edge-ccs into k-edge-subgraphs. The running time is more than $O(|V|)$.

For single values of k it is faster to use `nx.k_edge_subgraphs`. But for multiple values of k, it can be faster to build AuxGraph and then use this method.

3.16.3 K-node-components

Moody and White algorithm for k-components

k_components(G[, flow_func])	Returns the k-component structure of a graph G.

k_components

k_components(*G*, *flow_func=None*)

Returns the 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.

Parameters

G
[NetworkX graph]

flow_func
[function] Function to perform the underlying flow computations. Default value `edmonds_karp()`. This function performs better in sparse graphs with right tailed degree distributions. `shortest_augmenting_path()` will perform better in denser graphs.

Returns

k_components
: [dict] Dictionary with all connectivity levels `k` in the input Graph as keys and a list of sets of nodes that form a k-component of level `k` as values.

Raises

NetworkXNotImplemented
: If the input graph is directed.

See also:

node_connectivity
all_node_cuts
biconnected_components
: special case of this function when k=2

k_edge_components
: similar to this function, but uses edge-connectivity instead of node-connectivity

Notes

Moody and White [1] (appendix A) provide an algorithm for identifying k-components in a graph, which is based on Kanevsky's algorithm [2] for finding all minimum-size node cut-sets of a graph (implemented in `all_node_cuts()` function):

1. Compute node connectivity, k, of the input graph G.
2. Identify all k-cutsets at the current level of connectivity using Kanevsky's algorithm.
3. Generate new graph components based on the removal of these cutsets. Nodes in a cutset belong to both sides of the induced cut.
4. If the graph is neither complete nor trivial, return to 1; else end.

This implementation also uses some heuristics (see [3] for details) to speed up the computation.

References

[1], [2], [3]

Examples

```
>>> # Petersen graph has 10 nodes and it is triconnected, thus all
>>> # nodes are in a single component on all three connectivity levels
>>> G = nx.petersen_graph()
>>> k_components = nx.k_components(G)
```

3.16.4 K-node-cutsets

Kanevsky all minimum node k cutsets algorithm.

all_node_cuts(G[, k, flow_func])	Returns all minimum k cutsets of an undirected graph G.

all_node_cuts

all_node_cuts(*G*, *k=None*, *flow_func=None*)

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 *edmonds_karp()*. This function performs better in sparse graphs with right tailed degree distributions. *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.

See also:

node_connectivity
edmonds_karp
shortest_augmenting_path

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.

References

[1]

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
```

3.16.5 Flow-based disjoint paths

Flow based node and edge disjoint paths.

edge_disjoint_paths(G, s, t[, flow_func, ...])	Returns the edges disjoint paths between source and target.
node_disjoint_paths(G, s, t[, flow_func, ...])	Computes node disjoint paths between source and target.

edge_disjoint_paths

edge_disjoint_paths(*G*, *s*, *t*, *flow_func=None*, *cutoff=None*, *auxiliary=None*, *residual=None*)

Returns the edges disjoint paths between source and target.

Edge disjoint paths are paths that do not share any edge. The number of edge disjoint paths between source and target is equal to their edge connectivity.

Parameters

G
[NetworkX graph]

s
[node] Source node for the flow.

t
[node] Sink node for the flow.

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 `maximum_flow()` for details). If flow_func is None, the default maximum flow function (`edmonds_karp()`) is used. 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 edge 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] A generator of edge independent paths.

Raises

NetworkXNoPath
: If there is no path between source and target.

NetworkXError
: If source or target are not in the graph G.

See also:

node_disjoint_paths()
edge_connectivity()
maximum_flow()
edmonds_karp()
preflow_push()
shortest_augmenting_path()

Notes

This is a flow based implementation of edge disjoint paths. We compute the maximum flow between source and target on an auxiliary directed network. The saturated edges in the residual network after running the maximum flow algorithm correspond to edge disjoint paths between source and target in the original network. This function handles both directed and undirected graphs, and can use all flow algorithms from NetworkX flow package.

Examples

We use in this example the platonic icosahedral graph, which has node edge connectivity 5, thus there are 5 edge disjoint paths between any pair of nodes.

```
>>> G = nx.icosahedral_graph()
>>> len(list(nx.edge_disjoint_paths(G, 0, 6)))
5
```

If you need to compute edge disjoint paths on 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 edge connectivity, and the residual network for the underlying maximum flow computation.

Example of how to compute edge disjoint paths among all pairs of nodes of the platonic icosahedral graph reusing the data structures.

```
>>> import itertools
>>> # You also have to explicitly import the function for
>>> # building the auxiliary digraph from the connectivity package
>>> from networkx.algorithms.connectivity import build_auxiliary_edge_connectivity
>>> H = build_auxiliary_edge_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")
>>> result = {n: {} for n in G}
>>> # Reuse the auxiliary digraph and the residual network by passing them
>>> # as arguments
>>> for u, v in itertools.combinations(G, 2):
...     k = len(list(nx.edge_disjoint_paths(G, u, v, auxiliary=H, residual=R)))
...     result[u][v] = k
>>> all(result[u][v] == 5 for u, v in itertools.combinations(G, 2))
True
```

You can also use alternative flow algorithms for computing edge disjoint paths. For instance, in dense networks the algorithm `shortest_augmenting_path()` will usually perform better than the default `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
>>> len(list(nx.edge_disjoint_paths(G, 0, 6, flow_func=shortest_augmenting_path)))
5
```

node_disjoint_paths

node_disjoint_paths(*G*, *s*, *t*, *flow_func=None*, *cutoff=None*, *auxiliary=None*, *residual=None*)

Computes node disjoint paths between source and target.

Node disjoint paths are paths that only share their first and last nodes. The number of node independent paths between two nodes is equal to their local node connectivity.

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 `maximum_flow()` for details). If flow_func is None, the default maximum flow function (`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.

See also:

edge_disjoint_paths()
node_connectivity()
maximum_flow()
edmonds_karp()
preflow_push()
shortest_augmenting_path()

Notes

This is a flow based implementation of node disjoint paths. We compute the maximum flow between source and target on an auxiliary directed network. The saturated edges in the residual network after running the maximum flow algorithm correspond to node disjoint paths between source and target in the original network. This function handles both directed and undirected graphs, and can use all flow algorithms from NetworkX flow package.

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 `shortest_augmenting_path()` will usually perform better than the default `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
>>> len(list(nx.node_disjoint_paths(G, 0, 6, flow_func=shortest_augmenting_path)))
5
```

3.16.6 Flow-based Connectivity

Flow based connectivity algorithms

average_node_connectivity(G[, flow_func])	Returns the average connectivity of a graph G.
all_pairs_node_connectivity(G[, nbunch, ...])	Compute node connectivity between all pairs of nodes of G.
edge_connectivity(G[, s, t, flow_func, cutoff])	Returns the edge connectivity of the graph or digraph G.
local_edge_connectivity(G, s, t[, ...])	Returns local edge connectivity for nodes s and t in G.
local_node_connectivity(G, s, t[, ...])	Computes local node connectivity for nodes s and t.
node_connectivity(G[, s, t, flow_func])	Returns node connectivity for a graph or digraph G.

average_node_connectivity

average_node_connectivity(*G*, *flow_func=None*)

Returns the average connectivity of a graph G.

The average connectivity `bar{kappa}` of a graph G is the average of local node connectivity over all pairs of nodes of G [1] .

$$\bar{\kappa}(G) = \frac{\sum_{u,v} \kappa_G(u,v)}{\binom{n}{2}}$$

Parameters

G
[NetworkX graph] Undirected graph

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 `maximum_flow()` for details). If flow_func is None, the default maximum flow function (`edmonds_karp()`) is used. See *`local_node_connectivity()`* for details. The choice of the default function may change from version to version and should not be relied on. Default value: None.

Returns

K
[float] Average node connectivity

See also:

`local_node_connectivity()`
`node_connectivity()`
`edge_connectivity()`
`maximum_flow()`
`edmonds_karp()`
`preflow_push()`
`shortest_augmenting_path()`

References

[1]

all_pairs_node_connectivity

`all_pairs_node_connectivity`(*G*, *nbunch=None*, *flow_func=None*)

Compute node connectivity between all pairs of nodes of G.

Parameters

G
[NetworkX graph] Undirected graph

nbunch: container
Container of nodes. If provided node connectivity will be computed only over pairs of nodes in nbunch.

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 `maximum_flow()` for details). If flow_func is None, the default maximum flow function (`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.

Returns

all_pairs
[dict] A dictionary with node connectivity between all pairs of nodes in G, or in nbunch if provided.

See also:

local_node_connectivity()
edge_connectivity()
local_edge_connectivity()
maximum_flow()
edmonds_karp()
preflow_push()
shortest_augmenting_path()

edge_connectivity

edge_connectivity(*G*, *s=None*, *t=None*, *flow_func=None*, *cutoff=None*)

Returns the edge connectivity of the graph or digraph G.

The edge connectivity is equal to the minimum number of edges that must be removed to disconnect G or render it trivial. If source and target nodes are provided, this function returns the local edge connectivity: the minimum number of edges that must be removed to break all paths from source to target in G.

Parameters

G
[NetworkX graph] Undirected or directed graph

s
[node] Source node. Optional. Default value: None.

t
[node] Target node. Optional. Default value: None.

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 `maximum_flow()` for details). If flow_func is None, the default maximum flow function (`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, float, or None (default: None)] 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.

Returns

K
[integer] Edge connectivity for G, or local edge connectivity if source and target were provided

See also:

local_edge_connectivity()
local_node_connectivity()
node_connectivity()
maximum_flow()
edmonds_karp()
preflow_push()
shortest_augmenting_path()
k_edge_components()
k_edge_subgraphs()

Notes

This is a flow based implementation of global edge connectivity. For undirected graphs the algorithm works by finding a 'small' dominating set of nodes of G (see algorithm 7 in [1]) and computing local maximum flow (see *local_edge_connectivity()*) 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 maximum flow function. This is an implementation of algorithm 8 in [1] .

References

[1]

Examples

```
>>> # Platonic icosahedral graph is 5-edge-connected
>>> G = nx.icosahedral_graph()
>>> nx.edge_connectivity(G)
5
```

You can use alternative flow algorithms for the underlying maximum flow computation. In dense networks the algorithm `shortest_augmenting_path()` will usually perform better than the default `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
>>> nx.edge_connectivity(G, flow_func=shortest_augmenting_path)
5
```

If you specify a pair of nodes (source and target) as parameters, this function returns the value of local edge connectivity.

```
>>> nx.edge_connectivity(G, 3, 7)
5
```

If you need to perform several local computations 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 *local_edge_connectivity()* for details.

local_edge_connectivity

local_edge_connectivity(*G*, *s*, *t*, *flow_func=None*, *auxiliary=None*, *residual=None*, *cutoff=None*)

Returns local edge connectivity for nodes s and t in G.

Local edge connectivity for two nodes s and t is the minimum number of edges that must be removed to disconnect them.

This is a flow based implementation of edge connectivity. We compute the maximum flow on an auxiliary digraph build from the original network (see below for details). This is equal to the local edge connectivity because the value of a maximum s-t-flow is equal to the capacity of a minimum s-t-cut (Ford and Fulkerson theorem) [1] .

Parameters

G

[NetworkX graph] Undirected or directed 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 `maximum_flow()` for details). If flow_func is None, the default maximum flow function (`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.

auxiliary
[NetworkX DiGraph] Auxiliary digraph for computing flow based edge connectivity. 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.

cutoff
[integer, float, or None (default: None)] 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.

Returns

K
[integer] local edge connectivity for nodes s and t.

See also:

edge_connectivity()
local_node_connectivity()
node_connectivity()
maximum_flow()
edmonds_karp()
preflow_push()
shortest_augmenting_path()

Notes

This is a flow based implementation of edge connectivity. We compute the maximum flow using, by default, the `edmonds_karp()` algorithm on an auxiliary digraph build from the original input graph:

If the input graph is undirected, we replace each edge (`u`,`v`) with two reciprocal arcs (`u`, `v`) and (`v`, `u`) and then we set the attribute 'capacity' for each arc to 1. If the input graph is directed we simply add the 'capacity' attribute. This is an implementation of algorithm 1 in [1].

The maximum flow in the auxiliary network is equal to the local edge connectivity because the value of a maximum s-t-flow is equal to the capacity of a minimum s-t-cut (Ford and Fulkerson theorem).

References

[1]

Examples

This function is not imported in the base NetworkX namespace, so you have to explicitly import it from the connectivity package:

```
>>> from networkx.algorithms.connectivity import local_edge_connectivity
```

We use in this example the platonic icosahedral graph, which has edge connectivity 5.

```
>>> G = nx.icosahedral_graph()
>>> local_edge_connectivity(G, 0, 6)
5
```

If you need to compute local connectivity on 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 edge connectivity, and the residual network for the underlying maximum flow computation.

Example of how to compute local edge connectivity among all pairs of nodes of the platonic icosahedral graph reusing the data structures.

```
>>> import itertools
>>> # You also have to explicitly import the function for
>>> # building the auxiliary digraph from the connectivity package
>>> from networkx.algorithms.connectivity import build_auxiliary_edge_connectivity
>>> H = build_auxiliary_edge_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")
>>> result = dict.fromkeys(G, dict())
>>> # Reuse the auxiliary digraph and the residual network by passing them
>>> # as parameters
>>> for u, v in itertools.combinations(G, 2):
...     k = local_edge_connectivity(G, u, v, auxiliary=H, residual=R)
...     result[u][v] = k
>>> all(result[u][v] == 5 for u, v in itertools.combinations(G, 2))
True
```

You can also use alternative flow algorithms for computing edge connectivity. For instance, in dense networks the algorithm `shortest_augmenting_path()` will usually perform better than the default `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
>>> local_edge_connectivity(G, 0, 6, flow_func=shortest_augmenting_path)
5
```

local_node_connectivity

local_node_connectivity(*G*, *s*, *t*, *flow_func=None*, *auxiliary=None*, *residual=None*, *cutoff=None*)

Computes local node connectivity for nodes s and t.

Local node connectivity for two non adjacent nodes s and t is the minimum number of nodes that must be removed (along with their incident edges) to disconnect them.

This is a flow based implementation of node connectivity. We compute the maximum flow on an auxiliary digraph build from the original input graph (see below for details).

Parameters

G
[NetworkX graph] Undirected 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 `maximum_flow()` for details). If flow_func is None, the default maximum flow function (`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.

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.

cutoff
[integer, float, or None (default: None)] 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.

Returns

K
[integer] local node connectivity for nodes s and t

See also:

local_edge_connectivity()
node_connectivity()
minimum_node_cut()
maximum_flow()
edmonds_karp()
preflow_push()
shortest_augmenting_path()

Notes

This is a flow based implementation of node connectivity. We compute the maximum flow using, by default, the `edmonds_karp()` algorithm (see: `maximum_flow()`) on an auxiliary digraph build from the original input graph:

For an undirected graph G having `n` nodes and `m` edges we derive a directed graph H with `2n` nodes and `2m+n` arcs by replacing each original node `v` with two nodes `v_A`, `v_B` linked by an (internal) arc in H. Then for each edge (`u`, `v`) in G we add two arcs (`u_B`, `v_A`) and (`v_B`, `u_A`) in H. Finally we set the attribute capacity = 1 for each arc in H [1] .

For a directed graph G having `n` nodes and `m` arcs we derive a directed graph H with `2n` nodes and `m+n` arcs by replacing each original node `v` with two nodes `v_A`, `v_B` linked by an (internal) arc (`v_A`, `v_B`) in H. Then for each arc (`u`, `v`) in G we add one arc (`u_B`, `v_A`) in H. Finally we set the attribute capacity = 1 for each arc in H.

This is equal to the local node connectivity because the value of a maximum s-t-flow is equal to the capacity of a minimum s-t-cut.

References

[1]

Examples

This function is not imported in the base NetworkX namespace, so you have to explicitly import it from the connectivity package:

```
>>> from networkx.algorithms.connectivity import local_node_connectivity
```

We use in this example the platonic icosahedral graph, which has node connectivity 5.

```
>>> G = nx.icosahedral_graph()
>>> local_node_connectivity(G, 0, 6)
5
```

If you need to compute local connectivity on 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 the residual network for the underlying maximum flow computation.

Example of how to compute local node connectivity among all pairs of nodes of the platonic icosahedral graph reusing the data structures.

```
>>> import itertools
>>> # 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")
>>> result = dict.fromkeys(G, dict())
>>> # Reuse the auxiliary digraph and the residual network by passing them
>>> # as parameters
```

(continues on next page)

(continued from previous page)

```
>>> for u, v in itertools.combinations(G, 2):
...     k = local_node_connectivity(G, u, v, auxiliary=H, residual=R)
...     result[u][v] = k
...
>>> all(result[u][v] == 5 for u, v in itertools.combinations(G, 2))
True
```

You can also use alternative flow algorithms for computing node connectivity. For instance, in dense networks the algorithm `shortest_augmenting_path()` will usually perform better than the default `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
>>> local_node_connectivity(G, 0, 6, flow_func=shortest_augmenting_path)
5
```

node_connectivity

node_connectivity(*G*, *s=None*, *t=None*, *flow_func=None*)

Returns node connectivity for a graph or digraph G.

Node connectivity is equal to the minimum number of nodes that must be removed to disconnect G or render it trivial. If source and target nodes are provided, this function returns the local node connectivity: the minimum number of nodes that must be removed to break all paths from source to target in G.

Parameters

G
: [NetworkX graph] Undirected graph

s
: [node] Source node. Optional. Default value: None.

t
: [node] Target node. Optional. Default value: None.

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 `maximum_flow()` for details). If flow_func is None, the default maximum flow function (`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.

Returns

K
: [integer] Node connectivity of G, or local node connectivity if source and target are provided.

See also:

local_node_connectivity()
edge_connectivity()
maximum_flow()
edmonds_karp()
preflow_push()
shortest_augmenting_path()

Notes

This is a flow based implementation of node connectivity. The algorithm works by solving $O((n-\delta-1+\delta(\delta-1)/2))$ maximum flow problems on an auxiliary digraph. Where δ is the minimum degree of G. For details about the auxiliary digraph and the computation of local node connectivity see *local_node_connectivity()*. This implementation is based on algorithm 11 in [1].

References

[1]

Examples

```
>>> # Platonic icosahedral graph is 5-node-connected
>>> G = nx.icosahedral_graph()
>>> nx.node_connectivity(G)
5
```

You can use alternative flow algorithms for the underlying maximum flow computation. In dense networks the algorithm `shortest_augmenting_path()` will usually perform better than the default `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
>>> nx.node_connectivity(G, flow_func=shortest_augmenting_path)
5
```

If you specify a pair of nodes (source and target) as parameters, this function returns the value of local node connectivity.

```
>>> nx.node_connectivity(G, 3, 7)
5
```

If you need to perform several local computations 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 *local_node_connectivity()* for details.

3.16.7 Flow-based Minimum Cuts

Flow based cut algorithms

minimum_edge_cut(G[, s, t, flow_func])	Returns a set of edges of minimum cardinality that disconnects G.
minimum_node_cut(G[, s, t, flow_func])	Returns a set of nodes of minimum cardinality that disconnects G.
minimum_st_edge_cut(G, s, t[, flow_func, ...])	Returns the edges of the cut-set of a minimum (s, t)-cut.
minimum_st_node_cut(G, s, t[, flow_func, ...])	Returns a set of nodes of minimum cardinality that disconnect source from target in G.

minimum_edge_cut

minimum_edge_cut(*G*, *s=None*, *t=None*, *flow_func=None*)

Returns a set of edges of minimum cardinality that disconnects G.

If source and target nodes are provided, this function returns the set of edges of minimum cardinality that, if removed, would break all paths among source and target in G. If not, it returns a set of edges of minimum cardinality that disconnects G.

Parameters

G
[NetworkX graph]

s
[node] Source node. Optional. Default value: None.

t
[node] Target node. Optional. Default value: None.

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 `maximum_flow()` for details). If flow_func is None, the default maximum flow function (`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.

Returns

cutset
[set] Set of edges that, if removed, would disconnect G. If source and target nodes are provided, the set contains the edges that if removed, would destroy all paths between source and target.

See also:

minimum_st_edge_cut()
minimum_node_cut()
stoer_wagner()
node_connectivity()
edge_connectivity()
maximum_flow()
edmonds_karp()
preflow_push()
shortest_augmenting_path()

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].

References

[1]

Examples

```
>>> # Platonic icosahedral graph has edge connectivity 5
>>> G = nx.icosahedral_graph()
>>> len(nx.minimum_edge_cut(G))
5
```

You can use alternative flow algorithms for the underlying maximum flow computation. In dense networks the algorithm `shortest_augmenting_path()` will usually perform better than the default `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
>>> len(nx.minimum_edge_cut(G, flow_func=shortest_augmenting_path))
5
```

If you specify a pair of nodes (source and target) as parameters, this function returns the value of local edge connectivity.

```
>>> nx.edge_connectivity(G, 3, 7)
5
```

If you need to perform several local computations 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 `local_edge_connectivity()` for details.

minimum_node_cut

minimum_node_cut(*G*, *s=None*, *t=None*, *flow_func=None*)

Returns a set of nodes of minimum cardinality that disconnects G.

If source and target nodes are provided, this function returns the set of nodes of minimum cardinality that, if removed, would destroy all paths among source and target in G. If not, it returns a set of nodes of minimum cardinality that disconnects G.

Parameters

G
[NetworkX graph]

s
[node] Source node. Optional. Default value: None.

t
[node] Target node. Optional. Default value: None.

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 `maximum_flow()` for details). If flow_func is None, the default maximum flow function (`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.

Returns

cutset
[set] Set of nodes that, if removed, would disconnect G. If source and target nodes are provided, the set contains the nodes that if removed, would destroy all paths between source and target.

See also:

minimum_st_node_cut()
minimum_cut()
minimum_edge_cut()
stoer_wagner()
node_connectivity()
edge_connectivity()
maximum_flow()
edmonds_karp()
preflow_push()
shortest_augmenting_path()

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].

References

[1]

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 `shortest_augmenting_path()` will usually perform better than the default `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 *minimum_st_node_cut()* for details.

minimum_st_edge_cut

minimum_st_edge_cut(*G*, *s*, *t*, *flow_func=None*, *auxiliary=None*, *residual=None*)

Returns the edges of the cut-set of a minimum (s, t)-cut.

This function returns the set of edges of minimum cardinality that, if removed, would destroy all paths among source and target in G. Edge weights are not considered. See `minimum_cut()` for computing minimum cuts considering edge weights.

Parameters

G
[NetworkX graph]

s
[node] Source node for the flow.

t
[node] Sink node for the flow.

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.

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 `maximum_flow()` for details). If flow_func is None, the default maximum flow function (`edmonds_karp()`) is used. See `node_connectivity()` for details. The choice of the default function may change from version to version and should not be relied on. 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

cutset
[set] Set of edges that, if removed from the graph, will disconnect it.

See also:

minimum_cut()
minimum_node_cut()
minimum_edge_cut()
stoer_wagner()
node_connectivity()
edge_connectivity()
maximum_flow()
edmonds_karp()
preflow_push()
shortest_augmenting_path()

Examples

This function is not imported in the base NetworkX namespace, so you have to explicitly import it from the connectivity package:

```
>>> from networkx.algorithms.connectivity import minimum_st_edge_cut
```

We use in this example the platonic icosahedral graph, which has edge connectivity 5.

```
>>> G = nx.icosahedral_graph()
>>> len(minimum_st_edge_cut(G, 0, 6))
5
```

If you need to compute local edge cuts on 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 edge connectivity, and the residual network for the underlying maximum flow computation.

Example of how to compute local edge cuts among all pairs of nodes of the platonic icosahedral graph reusing the data structures.

```
>>> import itertools
>>> # You also have to explicitly import the function for
>>> # building the auxiliary digraph from the connectivity package
>>> from networkx.algorithms.connectivity import build_auxiliary_edge_connectivity
>>> H = build_auxiliary_edge_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")
>>> result = dict.fromkeys(G, dict())
>>> # Reuse the auxiliary digraph and the residual network by passing them
>>> # as parameters
>>> for u, v in itertools.combinations(G, 2):
...     k = len(minimum_st_edge_cut(G, u, v, auxiliary=H, residual=R))
...     result[u][v] = k
>>> all(result[u][v] == 5 for u, v in itertools.combinations(G, 2))
True
```

You can also use alternative flow algorithms for computing edge cuts. For instance, in dense networks the algorithm `shortest_augmenting_path()` will usually perform better than the default `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
>>> len(minimum_st_edge_cut(G, 0, 6, flow_func=shortest_augmenting_path))
5
```

minimum_st_node_cut

minimum_st_node_cut(*G*, *s*, *t*, *flow_func=None*, *auxiliary=None*, *residual=None*)

Returns a set of nodes of minimum cardinality that disconnect source from target in G.

This function returns the set of nodes of minimum cardinality that, if removed, would destroy all paths among source and target in G.

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 `maximum_flow()` for details). If flow_func is None, the default maximum flow function (`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.

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

cutset
[set] Set of nodes that, if removed, would destroy all paths between source and target in G.

See also:

minimum_node_cut()
minimum_edge_cut()
stoer_wagner()
node_connectivity()
edge_connectivity()
maximum_flow()
edmonds_karp()
preflow_push()
shortest_augmenting_path()

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].

References

[1]

Examples

This function is not imported in the base NetworkX namespace, so you have to explicitly import it from the connectivity package:

```
>>> from networkx.algorithms.connectivity import minimum_st_node_cut
```

We use in this example the platonic icosahedral graph, which has node connectivity 5.

```
>>> G = nx.icosahedral_graph()
>>> len(minimum_st_node_cut(G, 0, 6))
5
```

If you need to compute local st cuts 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 local st node cuts 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 parameters
>>> len(minimum_st_node_cut(G, 0, 6, auxiliary=H, residual=R))
5
```

You can also use alternative flow algorithms for computing minimum st node cuts. For instance, in dense networks the algorithm `shortest_augmenting_path()` will usually perform better than the default `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
>>> len(minimum_st_node_cut(G, 0, 6, flow_func=shortest_augmenting_path))
5
```

3.16.8 Stoer-Wagner minimum cut

Stoer-Wagner minimum cut algorithm.

stoer_wagner(G[, weight, heap])	Returns the weighted minimum edge cut using the Stoer-Wagner algorithm.

stoer_wagner

stoer_wagner(*G*, *weight='weight'*, *heap=<class 'networkx.utils.heaps.BinaryHeap'>*)

Returns the weighted minimum edge cut using the Stoer-Wagner algorithm.

Determine the minimum edge cut of a connected graph using the Stoer-Wagner algorithm. In weighted cases, all weights must be nonnegative.

The running time of the algorithm depends on the type of heaps used:

Type of heap	Running time
Binary heap	$O(n(m+n)\log n)$
Fibonacci heap	$O(nm + n^2 \log n)$
Pairing heap	$O(2^{2\sqrt{\log\log n}} nm + n^2 \log n)$

Parameters

G
[NetworkX graph] Edges of the graph are expected to have an attribute named by the weight parameter below. If this attribute is not present, the edge is considered to have unit weight.

weight
[string] Name of the weight attribute of the edges. If the attribute is not present, unit weight is assumed. Default value: 'weight'.

heap
[class] Type of heap to be used in the algorithm. It should be a subclass of `MinHeap` or implement a compatible interface.

If a stock heap implementation is to be used, `BinaryHeap` is recommended over `PairingHeap` for Python implementations without optimized attribute accesses (e.g., CPython) despite a slower asymptotic running time. For Python implementations with optimized attribute accesses (e.g., PyPy), `PairingHeap` provides better performance. Default value: `BinaryHeap`.

Returns

cut_value
[integer or float] The sum of weights of edges in a minimum cut.

partition
[pair of node lists] A partitioning of the nodes that defines a minimum cut.

Raises

NetworkXNotImplemented
If the graph is directed or a multigraph.

NetworkXError
If the graph has less than two nodes, is not connected or has a negative-weighted edge.

Examples

```
>>> G = nx.Graph()
>>> G.add_edge("x", "a", weight=3)
>>> G.add_edge("x", "b", weight=1)
>>> G.add_edge("a", "c", weight=3)
>>> G.add_edge("b", "c", weight=5)
>>> G.add_edge("b", "d", weight=4)
>>> G.add_edge("d", "e", weight=2)
>>> G.add_edge("c", "y", weight=2)
>>> G.add_edge("e", "y", weight=3)
>>> cut_value, partition = nx.stoer_wagner(G)
>>> cut_value
4
```

3.16.9 Utils for flow-based connectivity

Utilities for connectivity package

build_auxiliary_edge_connectivity(G)	Auxiliary digraph for computing flow based edge connectivity
build_auxiliary_node_connectivity(G)	Creates a directed graph D from an undirected graph G to compute flow based node connectivity.

build_auxiliary_edge_connectivity

build_auxiliary_edge_connectivity(*G*)

Auxiliary digraph for computing flow based edge connectivity

If the input graph is undirected, we replace each edge (u,`v`) with two reciprocal arcs (u, v) and (v, u) and then we set the attribute 'capacity' for each arc to 1. If the input graph is directed we simply add the 'capacity' attribute. Part of algorithm 1 in [1] .

References

[1]

build_auxiliary_node_connectivity

build_auxiliary_node_connectivity(*G*)

Creates a directed graph D from an undirected graph G to compute flow based node connectivity.

For an undirected graph G having n nodes and m edges we derive a directed graph D with 2n nodes and 2m+n arcs by replacing each original node v with two nodes vA, vB linked by an (internal) arc in D. Then for each edge (u, v) in G we add two arcs (uB, vA) and (vB, uA) in D. Finally we set the attribute capacity = 1 for each arc in D [1].

For a directed graph having n nodes and m arcs we derive a directed graph D with 2n nodes and m+n arcs by replacing each original node v with two nodes vA, vB linked by an (internal) arc (vA, vB) in D. Then for each arc (u, v) in G we add one arc (uB, vA) in D. Finally we set the attribute capacity = 1 for each arc in D.

A dictionary with a mapping between nodes in the original graph and the auxiliary digraph is stored as a graph attribute: H.graph['mapping'].

References

[1]

3.17 Cores

Find the k-cores of a graph.

The k-core is found by recursively pruning nodes with degrees less than k.

See the following references for details:

An O(m) Algorithm for Cores Decomposition of Networks Vladimir Batagelj and Matjaz Zaversnik, 2003. https://arxiv.org/abs/cs.DS/0310049

Generalized Cores Vladimir Batagelj and Matjaz Zaversnik, 2002. https://arxiv.org/pdf/cs/0202039

For directed graphs a more general notion is that of D-cores which looks at (k, l) restrictions on (in, out) degree. The (k, k) D-core is the k-core.

D-cores: Measuring Collaboration of Directed Graphs Based on Degeneracy Christos Giatsidis, Dimitrios M. Thilikos, Michalis Vazirgiannis, ICDM 2011. http://www.graphdegeneracy.org/dcores_ICDM_2011.pdf

Multi-scale structure and topological anomaly detection via a new network statistic: The onion decomposition L. Hébert-Dufresne, J. A. Grochow, and A. Allard Scientific Reports 6, 31708 (2016) http://doi.org/10.1038/srep31708

core_number(G)	Returns the core number for each vertex.
k_core(G[, k, core_number])	Returns the k-core of G.
k_shell(G[, k, core_number])	Returns the k-shell of G.
k_crust(G[, k, core_number])	Returns the k-crust of G.
k_corona(G, k[, core_number])	Returns the k-corona of G.
k_truss(G, k)	Returns the k-truss of G.
onion_layers(G)	Returns the layer of each vertex in an onion decomposition of the graph.

3.17.1 core_number

core_number(*G*)

Returns the core number for each vertex.

A k-core is a maximal subgraph that contains nodes of degree k or more.

The core number of a node is the largest value k of a k-core containing that node.

Parameters

G
[NetworkX graph] A graph or directed graph

Returns

core_number
[dictionary] A dictionary keyed by node to the core number.

Raises

NetworkXError
The k-core is not implemented for graphs with self loops or parallel edges.

Notes

Not implemented for graphs with parallel edges or self loops.

For directed graphs the node degree is defined to be the in-degree + out-degree.

References

[1]

3.17.2 k_core

k_core(*G*, *k=None*, *core_number=None*)

Returns the k-core of G.

A k-core is a maximal subgraph that contains nodes of degree k or more.

Parameters

G
[NetworkX graph] A graph or directed graph

k
[int, optional] The order of the core. If not specified return the main core.

core_number
[dictionary, optional] Precomputed core numbers for the graph G.

Returns

G
[NetworkX graph] The k-core subgraph

Raises

NetworkXError
The k-core is not defined for graphs with self loops or parallel edges.

See also:

core_number

Notes

The main core is the core with the largest degree.

Not implemented for graphs with parallel edges or self loops.

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.

References

[1]

3.17.3 k_shell

k_shell(*G*, *k=None*, *core_number=None*)

Returns the k-shell of G.

The k-shell is the subgraph induced by nodes with core number k. That is, nodes in the k-core that are not in the (k+1)-core.

Parameters

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

NetworkXError
The k-shell is not implemented for graphs with self loops or parallel edges.

See also:

core_number
k_corona

Notes

This is similar to k_corona but in that case only neighbors in the k-core are considered.

Not implemented for graphs with parallel edges or self loops.

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.

References

[1]

3.17.4 k_crust

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.

Parameters

G
[NetworkX graph] A graph or directed graph.

k
[int, optional] The order of the shell. If not specified return the main crust.

core_number
[dictionary, optional] Precomputed core numbers for the graph G.

Returns

G
[NetworkX graph] The k-crust subgraph

Raises

NetworkXError
The k-crust is not implemented for graphs with self loops or parallel edges.

See also:

core_number

Notes

This definition of k-crust is different than the definition in [1]. The k-crust in [1] is equivalent to the k+1 crust of this algorithm.

Not implemented for graphs with parallel edges or self loops.

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.

References

[1]

3.17.5 k_corona

k_corona(*G*, *k*, *core_number=None*)

Returns the k-corona of G.

The k-corona is the subgraph of nodes in the k-core which have exactly k neighbours in the k-core.

Parameters

G
[NetworkX graph] A graph or directed graph

k
[int] The order of the corona.

core_number
[dictionary, optional] Precomputed core numbers for the graph G.

Returns

G
[NetworkX graph] The k-corona subgraph

Raises

NetworkXError
The k-cornoa is not defined for graphs with self loops or parallel edges.

See also:

core_number

Notes

Not implemented for graphs with parallel edges or self loops.

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.

References

[1]

3.17.6 k_truss

k_truss(*G*, *k*)

Returns the k-truss of `G`.

The k-truss is the maximal induced subgraph of `G` which contains at least three vertices where every edge is incident to at least `k-2` triangles.

Parameters

G
[NetworkX graph] An undirected graph

k
[int] The order of the truss

Returns

H
[NetworkX graph] The k-truss subgraph

Raises

NetworkXError
The k-truss is not defined for graphs with self loops or parallel edges or directed graphs.

Notes

A k-clique is a (k-2)-truss and a k-truss is a (k+1)-core.

Not implemented for digraphs or graphs with parallel edges or self loops.

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.

References

[1], [2]

3.17.7 onion_layers

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] A simple graph without self loops or parallel edges

Returns

od_layers
[dictionary] A dictionary keyed by vertex to the onion layer. The layers are contiguous integers starting at 1.

Raises

NetworkXError
The onion decomposition is not implemented for graphs with self loops or parallel edges or for directed graphs.

See also:

core_number

Notes

Not implemented for graphs with parallel edges or self loops.

Not implemented for directed graphs.

References

[1], [2]

3.18 Covering

Functions related to graph covers.

min_edge_cover(G[, matching_algorithm])	Returns the min cardinality edge cover of the graph as a set of edges.
is_edge_cover(G, cover)	Decides whether a set of edges is a valid edge cover of the graph.

3.18.1 min_edge_cover

min_edge_cover(*G*, *matching_algorithm=None*)

Returns the min cardinality edge cover of the graph as a set of edges.

A smallest edge cover can be found in polynomial time by finding a maximum matching and extending it greedily so that all nodes are covered. This function follows that process. A maximum matching algorithm can be specified for the first step of the algorithm. The resulting set may return a set with one 2-tuple for each edge, (the usual case) or with both 2-tuples `(u, v)` and `(v, u)` for each edge. The latter is only done when a bipartite matching algorithm is specified as `matching_algorithm`.

Parameters

G
[NetworkX graph] An undirected graph.

matching_algorithm
[function] A function that returns a maximum cardinality matching for `G`. The function

must take one input, the graph G, and return either a set of edges (with only one direction for the pair of nodes) or a dictionary mapping each node to its mate. If not specified, *max_weight_matching()* is used. Common bipartite matching functions include *hopcroft_karp_matching()* or *eppstein_matching()*.

Returns

min_cover
[set] A set of the edges in a minimum edge cover in the form of tuples. It contains only one of the equivalent 2-tuples `(u, v)` and `(v, u)` for each edge. If a bipartite method is used to compute the matching, the returned set contains both the 2-tuples `(u, v)` and `(v, u)` for each edge of a minimum edge cover.

Notes

An edge cover of a graph is a set of edges such that every node of the graph is incident to at least one edge of the set. The minimum edge cover is an edge covering of smallest cardinality.

Due to its implementation, the worst-case running time of this algorithm is bounded by the worst-case running time of the function `matching_algorithm`.

Minimum edge cover for `G` can also be found using the `min_edge_covering` function in *networkx.algorithms.bipartite.covering* which is simply this function with a default matching algorithm of `hopcraft_karp_matching()`

Examples

```
>>> G = nx.Graph([(0, 1), (0, 2), (0, 3), (1, 2), (1, 3)])
>>> sorted(nx.min_edge_cover(G))
[(2, 1), (3, 0)]
```

3.18.2 is_edge_cover

is_edge_cover(*G*, *cover*)

Decides whether a set of edges is a valid edge cover of the graph.

Given a set of edges, whether it is an edge covering can be decided if we just check whether all nodes of the graph has an edge from the set, incident on it.

Parameters

G
[NetworkX graph] An undirected bipartite graph.

cover
[set] Set of edges to be checked.

Returns

bool
Whether the set of edges is a valid edge cover of the graph.

Notes

An edge cover of a graph is a set of edges such that every node of the graph is incident to at least one edge of the set.

Examples

```
>>> G = nx.Graph([(0, 1), (0, 2), (0, 3), (1, 2), (1, 3)])
>>> cover = {(2, 1), (3, 0)}
>>> nx.is_edge_cover(G, cover)
True
```

3.19 Cycles

cycle_basis(G[, root])	Returns a list of cycles which form a basis for cycles of G.
simple_cycles(G)	Find simple cycles (elementary circuits) of a directed graph.
recursive_simple_cycles(G)	Find simple cycles (elementary circuits) of a directed graph.
find_cycle(G[, source, orientation])	Returns a cycle found via depth-first traversal.
minimum_cycle_basis(G[, weight])	Returns a minimum weight cycle basis for G

3.19.1 cycle_basis

cycle_basis(*G*, *root=None*)

Returns a list of cycles which form a basis for cycles of G.

A basis for cycles of a network is a minimal collection of cycles such that any cycle in the network can be written as a sum of cycles in the basis. Here summation of cycles is defined as "exclusive or" of the edges. Cycle bases are useful, e.g. when deriving equations for electric circuits using Kirchhoff's Laws.

Parameters

G
[NetworkX Graph]

root
[node, optional] Specify starting node for basis.

Returns

A list of cycle lists. Each cycle list is a list of nodes which forms a cycle (loop) in G.

See also:

simple_cycles

Notes

This is adapted from algorithm CACM 491 [1].

References

[1]

Examples

```
>>> G = nx.Graph()
>>> nx.add_cycle(G, [0, 1, 2, 3])
>>> nx.add_cycle(G, [0, 3, 4, 5])
>>> print(nx.cycle_basis(G, 0))
[[3, 4, 5, 0], [1, 2, 3, 0]]
```

3.19.2 simple_cycles

simple_cycles(G)

Find simple cycles (elementary circuits) of a directed graph.

A `simple cycle`, or `elementary circuit`, is a closed path where no node appears twice. Two elementary circuits are distinct if they are not cyclic permutations of each other.

This is a nonrecursive, iterator/generator version of Johnson's algorithm [1]. There may be better algorithms for some cases [2] [3].

Parameters

G
[NetworkX DiGraph] A directed graph

Yields

list of nodes
Each cycle is represented by a list of nodes along the cycle.

See also:

cycle_basis

Notes

The implementation follows pp. 79-80 in [1].

The time complexity is $O((n+e)(c+1))$ for n nodes, e edges and c elementary circuits.

References

[1], [2], [3]

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]]
```

3.19.3 recursive_simple_cycles

recursive_simple_cycles(*G*)

Find simple cycles (elementary circuits) of a directed graph.

A `simple cycle`, or `elementary circuit`, is a closed path where no node appears twice. Two elementary circuits are distinct if they are not cyclic permutations of each other.

This version uses a recursive algorithm to build a list of cycles. You should probably use the iterator version called simple_cycles(). Warning: This recursive version uses lots of RAM! It appears in NetworkX for pedagogical value.

Parameters

G
[NetworkX DiGraph] A directed graph

Returns

A list of cycles, where each cycle is represented by a list of nodes along the cycle.
Example:

```
>>> edges = [(0, 0), (0, 1), (0, 2), (1, 2), (2, 0), (2, 1), (2, 2)]
    ..
```

```
>>> G = nx.DiGraph(edges)
    ..
```

```
>>> nx.recursive_simple_cycles(G)
    ..
```

[[0], [2], [0, 1, 2], [0, 2], [1, 2]]

See also:

simple_cycles*, *cycle_basis

Notes

The implementation follows pp. 79-80 in [1].

The time complexity is $O((n+e)(c+1))$ for n nodes, e edges and c elementary circuits.

References

[1]

3.19.4 find_cycle

find_cycle(*G*, *source=None*, *orientation=None*)

Returns a cycle found via depth-first traversal.

The cycle is a list of edges indicating the cyclic path. Orientation of directed edges is controlled by `orientation`.

Parameters

G
[graph] A directed/undirected graph/multigraph.

source
[node, list of nodes] The node from which the traversal begins. If None, then a source is chosen arbitrarily and repeatedly until all edges from each node in the graph are searched.

orientation
[None | 'original' | 'reverse' | 'ignore' (default: None)] For directed graphs and directed multigraphs, edge traversals need not respect the original orientation of the edges. When set to 'reverse' every edge is traversed in the reverse direction. When set to 'ignore', every edge is treated as undirected. When set to 'original', every edge is treated as directed. In all three cases, the yielded edge tuples add a last entry to indicate the direction in which that edge was traversed. If orientation is None, the yielded edge has no direction indicated. The direction is respected, but not reported.

Returns

edges
[directed edges] A list of directed edges indicating the path taken for the loop. If no cycle is found, then an exception is raised. For graphs, an edge is of the form `(u, v)` where `u` and `v` are the tail and head of the edge as determined by the traversal. For multigraphs, an edge is of the form `(u, v, key)`, where `key` is the key of the edge. When the graph is directed, then `u` and `v` are always in the order of the actual directed edge. If orientation is not None then the edge tuple is extended to include the direction of traversal ('forward' or 'reverse') on that edge.

Raises

NetworkXNoCycle
If no cycle was found.

See also:

simple_cycles

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')]
```

3.19.5 minimum_cycle_basis

minimum_cycle_basis(*G*, *weight=None*)

Returns a minimum weight cycle basis for G

Minimum weight means a cycle basis for which the total weight (length for unweighted graphs) of all the cycles is minimum.

Parameters

G
[NetworkX Graph]

weight: string
name of the edge attribute to use for edge weights

Returns

A list of cycle lists. Each cycle list is a list of nodes which forms a cycle (loop) in G. Note that the nodes are not necessarily returned in a order by which they appear in the cycle

See also:

simple_cycles*, *cycle_basis

Examples

```
>>> G = nx.Graph()
>>> nx.add_cycle(G, [0, 1, 2, 3])
>>> nx.add_cycle(G, [0, 3, 4, 5])
>>> print([sorted(c) for c in nx.minimum_cycle_basis(G)])
[[0, 1, 2, 3], [0, 3, 4, 5]]
```

References:

[1] Kavitha, Telikepalli, et al. "An O(m^2n) Algorithm for Minimum Cycle Basis of Graphs." http://link.springer.com/article/10.1007/s00453-007-9064-z [2] de Pina, J. 1995. Applications of shortest path methods. Ph.D. thesis, University of Amsterdam, Netherlands

3.20 Cuts

Functions for finding and evaluating cuts in a graph.

boundary_expansion(G, S)	Returns the boundary expansion of the set S.
conductance(G, S[, T, weight])	Returns the conductance of two sets of nodes.
cut_size(G, S[, T, weight])	Returns the size of the cut between two sets of nodes.
edge_expansion(G, S[, T, weight])	Returns the edge expansion between two node sets.
mixing_expansion(G, S[, T, weight])	Returns the mixing expansion between two node sets.
node_expansion(G, S)	Returns the node expansion of the set S.
normalized_cut_size(G, S[, T, weight])	Returns the normalized size of the cut between two sets of nodes.
volume(G, S[, weight])	Returns the volume of a set of nodes.

3.20.1 boundary_expansion

boundary_expansion(*G*, *S*)

Returns the boundary expansion of the set S.

The *boundary expansion* is the quotient of the size of the node boundary and the cardinality of *S*. [1]

Parameters

G
[NetworkX graph]

S
[collection] A collection of nodes in G.

Returns

number
The boundary expansion of the set S.

See also:

edge_expansion
mixing_expansion
node_expansion

References

[1]

3.20.2 conductance

conductance(*G*, *S*, *T=None*, *weight=None*)

Returns the conductance of two sets of nodes.

The *conductance* is the quotient of the cut size and the smaller of the volumes of the two sets. [1]

Parameters

G
[NetworkX graph]

S
[collection] A collection of nodes in `G`.

T
[collection] A collection of nodes in `G`.

weight
[object] Edge attribute key to use as weight. If not specified, edges have weight one.

Returns

number
The conductance between the two sets `S` and `T`.

See also:

cut_size
edge_expansion
normalized_cut_size
volume

References

[1]

3.20.3 cut_size

cut_size(*G*, *S*, *T=None*, *weight=None*)

Returns the size of the cut between two sets of nodes.

A *cut* is a partition of the nodes of a graph into two sets. The *cut size* is the sum of the weights of the edges "between" the two sets of nodes.

Parameters

G
[NetworkX graph]

S
[collection] A collection of nodes in `G`.

T
[collection] A collection of nodes in `G`. If not specified, this is taken to be the set complement of `S`.

weight
[object] Edge attribute key to use as weight. If not specified, edges have weight one.

Returns

number
Total weight of all edges from nodes in set `S` to nodes in set `T` (and, in the case of directed graphs, all edges from nodes in `T` to nodes in `S`).

Notes

In a multigraph, the cut size is the total weight of edges including multiplicity.

Examples

In the graph with two cliques joined by a single edges, the natural bipartition of the graph into two blocks, one for each clique, yields a cut of weight one:

```
>>> G = nx.barbell_graph(3, 0)
>>> S = {0, 1, 2}
>>> T = {3, 4, 5}
>>> nx.cut_size(G, S, T)
1
```

Each parallel edge in a multigraph is counted when determining the cut size:

```
>>> G = nx.MultiGraph(["ab", "ab"])
>>> S = {"a"}
>>> T = {"b"}
>>> nx.cut_size(G, S, T)
2
```

3.20.4 edge_expansion

edge_expansion(*G*, *S*, *T=None*, *weight=None*)

Returns the edge expansion between two node sets.

The *edge expansion* is the quotient of the cut size and the smaller of the cardinalities of the two sets. [1]

Parameters

G
[NetworkX graph]

S
[collection] A collection of nodes in `G`.

T
[collection] A collection of nodes in `G`.

weight
[object] Edge attribute key to use as weight. If not specified, edges have weight one.

Returns

number
The edge expansion between the two sets `S` and `T`.

See also:

boundary_expansion
mixing_expansion
node_expansion

References

[1]

3.20.5 mixing_expansion

mixing_expansion(*G*, *S*, *T=None*, *weight=None*)

Returns the mixing expansion between two node sets.

The *mixing expansion* is the quotient of the cut size and twice the number of edges in the graph. [1]

Parameters

G
[NetworkX graph]

S
[collection] A collection of nodes in `G`.

T
[collection] A collection of nodes in `G`.

weight
[object] Edge attribute key to use as weight. If not specified, edges have weight one.

Returns

number
The mixing expansion between the two sets `S` and `T`.

See also:

boundary_expansion
edge_expansion
node_expansion

References

[1]

3.20.6 node_expansion

node_expansion(*G*, *S*)

Returns the node expansion of the set `S`.

The *node expansion* is the quotient of the size of the node boundary of *S* and the cardinality of *S*. [1]

Parameters

G
[NetworkX graph]

S
[collection] A collection of nodes in `G`.

Returns

number
The node expansion of the set `S`.

See also:

boundary_expansion
edge_expansion
mixing_expansion

References

[1]

3.20.7 normalized_cut_size

normalized_cut_size(*G*, *S*, *T=None*, *weight=None*)

Returns the normalized size of the cut between two sets of nodes.

The *normalized cut size* is the cut size times the sum of the reciprocal sizes of the volumes of the two sets. [1]

Parameters

G
[NetworkX graph]

S
[collection] A collection of nodes in `G`.

T
[collection] A collection of nodes in `G`.

weight
[object] Edge attribute key to use as weight. If not specified, edges have weight one.

Returns

number
The normalized cut size between the two sets `S` and `T`.

See also:

conductance
cut_size
edge_expansion
volume

Notes

In a multigraph, the cut size is the total weight of edges including multiplicity.

References

[1]

3.20.8 volume

volume(*G*, *S*, *weight=None*)

Returns the volume of a set of nodes.

The *volume* of a set *S* is the sum of the (out-)degrees of nodes in *S* (taking into account parallel edges in multigraphs). [1]

Parameters

G
[NetworkX graph]

S
[collection] A collection of nodes in `G`.

weight
[object] Edge attribute key to use as weight. If not specified, edges have weight one.

Returns

number
The volume of the set of nodes represented by `S` in the graph `G`.

See also:

conductance
cut_size
edge_expansion
edge_boundary
normalized_cut_size

References

[1]

3.21 D-Separation

Algorithm for testing d-separation in DAGs.

d-separation is a test for conditional independence in probability distributions that can be factorized using DAGs. It is a purely graphical test that uses the underlying graph and makes no reference to the actual distribution parameters. See [1] for a formal definition.

The implementation is based on the conceptually simple linear time algorithm presented in [2]. Refer to [3], [4] for a couple of alternative algorithms.

Here, we provide a brief overview of d-separation and related concepts that are relevant for understanding it:

3.21.1 Blocking paths

Before we overview, we introduce the following terminology to describe paths:

- "open" path: A path between two nodes that can be traversed
- "blocked" path: A path between two nodes that cannot be traversed

A **collider** is a triplet of nodes along a path that is like the following: `... u -> c <- v ...`), where 'c' is a common successor of `u` and `v`. A path through a collider is considered "blocked". When a node that is a collider, or a descendant of a collider is included in the d-separating set, then the path through that collider node is "open". If the path through the collider node is open, then we will call this node an open collider.

The d-separation set blocks the paths between `u` and `v`. If you include colliders, or their descendant nodes in the d-separation set, then those colliders will open up, enabling a path to be traversed if it is not blocked some other way.

3.21.2 Illustration of D-separation with examples

For a pair of two nodes, `u` and `v`, all paths are considered open if there is a path between `u` and `v` that is not blocked. That means, there is an open path between `u` and `v` that does not encounter a collider, or a variable in the d-separating set.

For example, if the d-separating set is the empty set, then the following paths are unblocked between `u` and `v`:

- u <- z -> v
- u -> w -> … -> z -> v

If for example, 'z' is in the d-separating set, then 'z' blocks those paths between `u` and `v`.

Colliders block a path by default if they and their descendants are not included in the d-separating set. An example of a path that is blocked when the d-separating set is empty is:

- u -> w -> … -> z <- v

because 'z' is a collider in this path and 'z' is not in the d-separating set. However, if 'z' or a descendant of 'z' is included in the d-separating set, then the path through the collider at 'z' (… -> z <- …) is now "open".

D-separation is concerned with blocking all paths between u and v. Therefore, a d-separating set between `u` and `v` is one where all paths are blocked.

3.21.3 D-separation and its applications in probability

D-separation is commonly used in probabilistic graphical models. D-separation connects the idea of probabilistic "dependence" with separation in a graph. If one assumes the causal Markov condition [5], then d-separation implies conditional independence in probability distributions.

3.21.4 Examples

```
>>>
>>> # HMM graph with five states and observation nodes
... g = nx.DiGraph()
>>> g.add_edges_from(
...     [
...         ("S1", "S2"),
...         ("S2", "S3"),
...         ("S3", "S4"),
...         ("S4", "S5"),
...         ("S1", "O1"),
...         ("S2", "O2"),
...         ("S3", "O3"),
...         ("S4", "O4"),
...         ("S5", "O5"),
...     ]
... )
>>>
>>> # states/obs before 'S3' are d-separated from states/obs after 'S3'
... nx.d_separated(g, {"S1", "S2", "O1", "O2"}, {"S4", "S5", "O4", "O5"}, {"S3"})
True
```

3.21.5 References

d_separated(G, x, y, z)	Return whether node sets x and y are d-separated by z.

3.21.6 d_separated

d_separated(*G*, *x*, *y*, *z*)

Return whether node sets x and y are d-separated by z.

Parameters

G
[graph] A NetworkX DAG.

x
[set] First set of nodes in G.

y
[set] Second set of nodes in G.

z
[set] Set of conditioning nodes in G. Can be empty set.

Returns

b
[bool] A boolean that is true if x is d-separated from y given z in G.

Raises

NetworkXError
The *d-separation* test is commonly used with directed graphical models which are acyclic. Accordingly, the algorithm raises a NetworkXError if the input graph is not a DAG.

NodeNotFound
If any of the input nodes are not found in the graph, a `NodeNotFound` exception is raised.

Notes

A d-separating set in a DAG is a set of nodes that blocks all paths between the two sets. Nodes in `z` block a path if they are part of the path and are not a collider, or a descendant of a collider. A collider structure along a path is `... -> c <- ...` where `c` is the collider node.

https://en.wikipedia.org/wiki/Bayesian_network#d-separation

3.22 Directed Acyclic Graphs

Algorithms for directed acyclic graphs (DAGs).

Note that most of these functions are only guaranteed to work for DAGs. In general, these functions do not check for acyclic-ness, so it is up to the user to check for that.

ancestors(G, source)	Returns all nodes having a path to `source` in `G`.
descendants(G, source)	Returns all nodes reachable from `source` in `G`.
topological_sort(G)	Returns a generator of nodes in topologically sorted order.
topological_generations(G)	Stratifies a DAG into generations.
all_topological_sorts(G)	Returns a generator of _all_ topological sorts of the directed graph G.
lexicographical_topological_sort(G[, key])	Generate the nodes in the unique lexicographical topological sort order.
is_directed_acyclic_graph(G)	Returns True if the graph `G` is a directed acyclic graph (DAG) or False if not.
is_aperiodic(G)	Returns True if `G` is aperiodic.
transitive_closure(G[, reflexive])	Returns transitive closure of a graph
transitive_closure_dag(G[, topo_order])	Returns the transitive closure of a directed acyclic graph.
transitive_reduction(G)	Returns transitive reduction of a directed graph
antichains(G[, topo_order])	Generates antichains from a directed acyclic graph (DAG).
dag_longest_path(G[, weight, ...])	Returns the longest path in a directed acyclic graph (DAG).
dag_longest_path_length(G[, weight, ...])	Returns the longest path length in a DAG
dag_to_branching(G)	Returns a branching representing all (overlapping) paths from root nodes to leaf nodes in the given directed acyclic graph.

3.22.1 ancestors

ancestors(*G*, *source*)

Returns all nodes having a path to `source` in `G`.

Parameters

G
[NetworkX Graph]

source
[node in `G`]

Returns

set()
The ancestors of `source` in `G`

Raises

NetworkXError
If node `source` is not in `G`.

See also:

descendants

Examples

```
>>> DG = nx.path_graph(5, create_using=nx.DiGraph)
>>> sorted(nx.ancestors(DG, 2))
[0, 1]
```

The `source` node is not an ancestor of itself, but can be included manually:

```
>>> sorted(nx.ancestors(DG, 2) | {2})
[0, 1, 2]
```

3.22.2 descendants

descendants(*G*, *source*)

Returns all nodes reachable from `source` in `G`.

Parameters

G
[NetworkX Graph]

source
[node in `G`]

Returns

set()
The descendants of `source` in `G`

Raises

NetworkXError
If node `source` is not in `G`.

See also:

ancestors

Examples

```
>>> DG = nx.path_graph(5, create_using=nx.DiGraph)
>>> sorted(nx.descendants(DG, 2))
[3, 4]
```

The `source` node is not a descendant of itself, but can be included manually:

```
>>> sorted(nx.descendants(DG, 2) | {2})
[2, 3, 4]
```

3.22.3 topological_sort

topological_sort(*G*)

Returns a generator of nodes in topologically sorted order.

A topological sort is a nonunique permutation of the nodes of a directed graph such that an edge from u to v implies that u appears before v in the topological sort order. This ordering is valid only if the graph has no directed cycles.

Parameters

G
[NetworkX digraph] A directed acyclic graph (DAG)

Yields

nodes
Yields the nodes in topological sorted order.

Raises

NetworkXError
Topological sort is defined for directed graphs only. If the graph `G` is undirected, a `NetworkXError` is raised.

NetworkXUnfeasible
If `G` is not a directed acyclic graph (DAG) no topological sort exists and a `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.

See also:

is_directed_acyclic_graph, *lexicographical_topological_sort*

Notes

This algorithm is based on a description and proof in "Introduction to Algorithms: A Creative Approach" [1] .

References

[1]

Examples

To get the reverse order of the topological sort:

```
>>> DG = nx.DiGraph([(1, 2), (2, 3)])
>>> list(reversed(list(nx.topological_sort(DG))))
[3, 2, 1]
```

If your DiGraph naturally has the edges representing tasks/inputs and nodes representing people/processes that initiate tasks, then topological_sort is not quite what you need. You will have to change the tasks to nodes with dependence reflected by edges. The result is a kind of topological sort of the edges. This can be done with `networkx.line_graph()` as follows:

```
>>> list(nx.topological_sort(nx.line_graph(DG)))
[(1, 2), (2, 3)]
```

3.22.4 topological_generations

topological_generations(*G*)

Stratifies a DAG into generations.

A topological generation is node collection in which ancestors of a node in each generation are guaranteed to be in a previous generation, and any descendants of a node are guaranteed to be in a following generation. Nodes are guaranteed to be in the earliest possible generation that they can belong to.

Parameters

G
[NetworkX digraph] A directed acyclic graph (DAG)

Yields

sets of nodes
Yields sets of nodes representing each generation.

Raises

NetworkXError
Generations are defined for directed graphs only. If the graph `G` is undirected, a `NetworkXError` is raised.

NetworkXUnfeasible
If `G` is not a directed acyclic graph (DAG) no topological generations exist and a `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.

See also:

topological_sort

Notes

The generation in which a node resides can also be determined by taking the max-path-distance from the node to the farthest leaf node. That value can be obtained with this function using `enumerate(topological_generations(G))`.

Examples

```
>>> DG = nx.DiGraph([(2, 1), (3, 1)])
>>> [sorted(generation) for generation in nx.topological_generations(DG)]
[[2, 3], [1]]
```

3.22.5 all_topological_sorts

all_topological_sorts(*G*)

Returns a generator of _all_ topological sorts of the directed graph G.

A topological sort is a nonunique permutation of the nodes such that an edge from u to v implies that u appears before v in the topological sort order.

Parameters

G
[NetworkX DiGraph] A directed graph

Yields

topological_sort_order
[list] a list of nodes in `G`, representing one of the topological sort orders

Raises

NetworkXNotImplemented
If `G` is not directed

NetworkXUnfeasible
If `G` is not acyclic

Notes

Implements an iterative version of the algorithm given in [1].

References

[1]

Examples

To enumerate all topological sorts of directed graph:

```
>>> DG = nx.DiGraph([(1, 2), (2, 3), (2, 4)])
>>> list(nx.all_topological_sorts(DG))
[[1, 2, 4, 3], [1, 2, 3, 4]]
```

3.22.6 lexicographical_topological_sort

lexicographical_topological_sort(*G*, *key=None*)

Generate the nodes in the unique lexicographical topological sort order.

Generates a unique ordering of nodes by first sorting topologically (for which there are often multiple valid orderings) and then additionally by sorting lexicographically.

A topological sort arranges the nodes of a directed graph so that the upstream node of each directed edge precedes the downstream node. It is always possible to find a solution for directed graphs that have no cycles. There may be more than one valid solution.

Lexicographical sorting is just sorting alphabetically. It is used here to break ties in the topological sort and to determine a single, unique ordering. This can be useful in comparing sort results.

The lexicographical order can be customized by providing a function to the `key=` parameter. The definition of the key function is the same as used in python's built-in `sort()`. The function takes a single argument and returns a key to use for sorting purposes.

Lexicographical sorting can fail if the node names are un-sortable. See the example below. The solution is to provide a function to the `key=` argument that returns sortable keys.

Parameters

G
[NetworkX digraph] A directed acyclic graph (DAG)

key
[function, optional] A function of one argument that converts a node name to a comparison key. It defines and resolves ambiguities in the sort order. Defaults to the identity function.

Yields

nodes
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 `NetworkXError` is raised.

NetworkXUnfeasible
If `G` is not a directed acyclic graph (DAG) no topological sort exists and a `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.

See also:

topological_sort

Notes

This algorithm is based on a description and proof in "Introduction to Algorithms: A Creative Approach" [1] .

References

[1]

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']
```

3.22.7 is_directed_acyclic_graph

is_directed_acyclic_graph(*G*)

Returns True if the graph G is a directed acyclic graph (DAG) or False if not.

Parameters

G
[NetworkX graph]

Returns

bool
True if G is a DAG, False otherwise

See also:

topological_sort

Examples

Undirected graph:

```
>>> G = nx.Graph([(1, 2), (2, 3)])
>>> nx.is_directed_acyclic_graph(G)
False
```

Directed graph with cycle:

```
>>> G = nx.DiGraph([(1, 2), (2, 3), (3, 1)])
>>> nx.is_directed_acyclic_graph(G)
False
```

Directed acyclic graph:

```
>>> G = nx.DiGraph([(1, 2), (2, 3)])
>>> nx.is_directed_acyclic_graph(G)
True
```

3.22.8 is_aperiodic

is_aperiodic(*G*)

Returns True if G is aperiodic.

A directed graph is aperiodic if there is no integer k > 1 that divides the length of every cycle in the graph.

Parameters

G
[NetworkX DiGraph] A directed graph

Returns

bool
True if the graph is aperiodic False otherwise

Raises

NetworkXError
If G is not directed

Notes

This uses the method outlined in [1], which runs in $O(m)$ time given m edges in G. Note that a graph is not aperiodic if it is acyclic as every integer trivial divides length 0 cycles.

References

[1]

Examples

A graph consisting of one cycle, the length of which is 2. Therefore `k = 2` divides the length of every cycle in the graph and thus the graph is *not aperiodic*:

```
>>> DG = nx.DiGraph([(1, 2), (2, 1)])
>>> nx.is_aperiodic(DG)
False
```

A graph consisting of two cycles: one of length 2 and the other of length 3. The cycle lengths are coprime, so there is no single value of k where `k > 1` that divides each cycle length and therefore the graph is *aperiodic*:

```
>>> DG = nx.DiGraph([(1, 2), (2, 3), (3, 1), (1, 4), (4, 1)])
>>> nx.is_aperiodic(DG)
True
```

A graph consisting of two cycles: one of length 2 and the other of length 4. The lengths of the cycles share a common factor `k = 2`, and therefore the graph is *not aperiodic*:

```
>>> DG = nx.DiGraph([(1, 2), (2, 1), (3, 4), (4, 5), (5, 6), (6, 3)])
>>> nx.is_aperiodic(DG)
False
```

An acyclic graph, therefore the graph is *not aperiodic*:

```
>>> DG = nx.DiGraph([(1, 2), (2, 3)])
>>> nx.is_aperiodic(DG)
False
```

3.22.9 transitive_closure

transitive_closure(*G*, *reflexive=False*)

Returns transitive closure of a graph

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 path from v to w in G.

Handling of paths from v to v has some flexibility within this definition. A reflexive transitive closure creates a self-loop for the path from v to v of length 0. The usual transitive closure creates a self-loop only if a cycle exists (a path from v to v with length > 0). We also allow an option for no self-loops.

Parameters

G
[NetworkX Graph] A directed/undirected graph/multigraph.

reflexive
[Bool or None, optional (default: False)] Determines when cycles create self-loops in the Transitive Closure. If True, trivial cycles (length 0) create self-loops. The result is a reflexive transitive closure of G. If False (the default) non-trivial cycles create self-loops. If None, self-loops are not created.

Returns

NetworkX graph
The transitive closure of `G`

Raises

NetworkXError
If `reflexive` not in `{None, True, False}`

References

[1]

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 then 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)])
```

3.22.10 transitive_closure_dag

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
The transitive closure of G

Raises

NetworkXNotImplemented
If G is not directed

NetworkXUnfeasible
If G has a cycle

Notes

This algorithm is probably simple enough to be well-known but I didn't find a mention in the literature.

Examples

```
>>> DG = nx.DiGraph([(1, 2), (2, 3)])
>>> TC = nx.transitive_closure_dag(DG)
>>> TC.edges()
OutEdgeView([(1, 2), (1, 3), (2, 3)])
```

3.22.11 transitive_reduction

transitive_reduction(*G*)

Returns transitive reduction of a directed graph

The transitive reduction 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 (v,w) is in E and there is no path from v to w in G with length greater than 1.

Parameters

G
[NetworkX DiGraph] A directed acyclic graph (DAG)

Returns

NetworkX DiGraph
The transitive reduction of `G`

Raises

NetworkXError
If `G` is not a directed acyclic graph (DAG) transitive reduction is not uniquely defined and a `NetworkXError` exception is raised.

References

https://en.wikipedia.org/wiki/Transitive_reduction

Examples

To perform transitive reduction on a DiGraph:

```
>>> DG = nx.DiGraph([(1, 2), (2, 3), (1, 3)])
>>> TR = nx.transitive_reduction(DG)
>>> list(TR.edges)
[(1, 2), (2, 3)]
```

To avoid unnecessary data copies, this implementation does not return a DiGraph with node/edge data. To perform transitive reduction on a DiGraph and transfer node/edge data:

```
>>> DG = nx.DiGraph()
>>> DG.add_edges_from([(1, 2), (2, 3), (1, 3)], color='red')
>>> TR = nx.transitive_reduction(DG)
>>> TR.add_nodes_from(DG.nodes(data=True))
>>> TR.add_edges_from((u, v, DG.edges[u, v]) for u, v in TR.edges)
>>> list(TR.edges(data=True))
[(1, 2, {'color': 'red'}), (2, 3, {'color': 'red'})]
```

3.22.12 antichains

antichains(*G*, *topo_order=None*)

Generates antichains from a directed acyclic graph (DAG).

An antichain is a subset of a partially ordered set such that any two elements in the subset are incomparable.

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)

Yields

antichain
[list] a list of nodes in `G` representing an antichain

Raises

NetworkXNotImplemented
If `G` is not directed

NetworkXUnfeasible
If `G` contains a cycle

Notes

This function was originally developed by Peter Jipsen and Franco Saliola for the SAGE project. It's included in NetworkX with permission from the authors. Original SAGE code at:

https://github.com/sagemath/sage/blob/master/src/sage/combinat/posets/hasse_diagram.py

References

[1]

Examples

```
>>> DG = nx.DiGraph([(1, 2), (1, 3)])
>>> list(nx.antichains(DG))
[[], [3], [2], [2, 3], [1]]
```

3.22.13 dag_longest_path

dag_longest_path(*G*, *weight='weight'*, *default_weight=1*, *topo_order=None*)

Returns the longest path in a directed acyclic graph (DAG).

If `G` has edges with `weight` attribute the edge data are used as weight values.

Parameters

G
[NetworkX DiGraph] A directed acyclic graph (DAG)

weight
[str, optional] Edge data key to use for weight

default_weight
[int, optional] The weight of edges that do not have a weight attribute

topo_order: list or tuple, optional
A topological order for `G` (if None, the function will compute one)

Returns

list
Longest path

Raises

NetworkXNotImplemented
If `G` is not directed

See also:

dag_longest_path_length

Examples

```
>>> DG = nx.DiGraph([(0, 1, {'cost':1}), (1, 2, {'cost':1}), (0, 2, {'cost':42})])
>>> list(nx.all_simple_paths(DG, 0, 2))
[[0, 1, 2], [0, 2]]
>>> nx.dag_longest_path(DG)
[0, 1, 2]
>>> nx.dag_longest_path(DG, weight="cost")
[0, 2]
```

In the case where multiple valid topological orderings exist, `topo_order` can be used to specify a specific ordering:

```
>>> DG = nx.DiGraph([(0, 1), (0, 2)])
>>> sorted(nx.all_topological_sorts(DG))  # Valid topological orderings
[[0, 1, 2], [0, 2, 1]]
>>> nx.dag_longest_path(DG, topo_order=[0, 1, 2])
[0, 1]
>>> nx.dag_longest_path(DG, topo_order=[0, 2, 1])
[0, 2]
```

3.22.14 dag_longest_path_length

dag_longest_path_length(*G*, *weight='weight'*, *default_weight=1*)

Returns the longest path length in a DAG

Parameters

G
[NetworkX DiGraph] A directed acyclic graph (DAG)

weight
[string, optional] Edge data key to use for weight

default_weight
[int, optional] The weight of edges that do not have a weight attribute

Returns

int
Longest path length

Raises

NetworkXNotImplemented
If `G` is not directed

See also:

dag_longest_path

Examples

```
>>> DG = nx.DiGraph([(0, 1, {'cost':1}), (1, 2, {'cost':1}), (0, 2, {'cost':42})])
>>> list(nx.all_simple_paths(DG, 0, 2))
[[0, 1, 2], [0, 2]]
>>> nx.dag_longest_path_length(DG)
2
>>> nx.dag_longest_path_length(DG, weight="cost")
42
```

3.22.15 dag_to_branching

dag_to_branching(*G*)

Returns a branching representing all (overlapping) paths from root nodes to leaf nodes in the given directed acyclic graph.

As described in *networkx.algorithms.tree.recognition*, a *branching* is a directed forest in which each node has at most one parent. In other words, a branching is a disjoint union of *arborescences*. For this function, each node of in-degree zero in G becomes a root of one of the arborescences, and there will be one leaf node for each distinct path from that root to a leaf node in G.

Each node v in G with *k* parents becomes *k* distinct nodes in the returned branching, one for each parent, and the sub-DAG rooted at v is duplicated for each copy. The algorithm then recurses on the children of each copy of v.

Parameters

G
[NetworkX graph] A directed acyclic graph.

Returns

DiGraph
The branching in which there is a bijection between root-to-leaf paths in G (in which multiple paths may share the same leaf) and root-to-leaf paths in the branching (in which there is a unique path from a root to a leaf).

Each node has an attribute 'source' whose value is the original node to which this node corresponds. No other graph, node, or edge attributes are copied into this new graph.

Raises

NetworkXNotImplemented
If G is not directed, or if G is a multigraph.

HasACycle
If G is not acyclic.

Notes

This function is not idempotent in the sense that the node labels in the returned branching may be uniquely generated each time the function is invoked. In fact, the node labels may not be integers; in order to relabel the nodes to be more readable, you can use the `networkx.convert_node_labels_to_integers()` function.

The current implementation of this function uses `networkx.prefix_tree()`, so it is subject to the limitations of that function.

Examples

To examine which nodes in the returned branching were produced by which original node in the directed acyclic graph, we can collect the mapping from source node to new nodes into a dictionary. For example, consider the directed diamond graph:

```
>>> from collections import defaultdict
>>> from operator import itemgetter
>>>
>>> G = nx.DiGraph(nx.utils.pairwise("abd"))
>>> G.add_edges_from(nx.utils.pairwise("acd"))
>>> B = nx.dag_to_branching(G)
>>>
>>> sources = defaultdict(set)
>>> for v, source in B.nodes(data="source"):
...     sources[source].add(v)
>>> len(sources["a"])
1
>>> len(sources["d"])
2
```

To copy node attributes from the original graph to the new graph, you can use a dictionary like the one constructed in the above example:

```
>>> for source, nodes in sources.items():
...     for v in nodes:
...         B.nodes[v].update(G.nodes[source])
```

3.23 Distance Measures

Graph diameter, radius, eccentricity and other properties.

barycenter(G[, weight, attr, sp])	Calculate barycenter of a connected graph, optionally with edge weights.
center(G[, e, usebounds, weight])	Returns the center of the graph G.
diameter(G[, e, usebounds, weight])	Returns the diameter of the graph G.
eccentricity(G[, v, sp, weight])	Returns the eccentricity of nodes in G.
periphery(G[, e, usebounds, weight])	Returns the periphery of the graph G.
radius(G[, e, usebounds, weight])	Returns the radius of the graph G.
resistance_distance(G, nodeA, nodeB[, ...])	Returns the resistance distance between node A and node B on graph G.

3.23.1 barycenter

barycenter(*G*, *weight=None*, *attr=None*, *sp=None*)

Calculate barycenter of a connected graph, optionally with edge weights.

The *barycenter* a `connected` graph G is the subgraph induced by the set of its nodes v minimizing the objective function

$$\sum_{u \in V(G)} d_G(u, v),$$

where d_G is the (possibly weighted) `path length`. The barycenter is also called the *median*. See [**?**], p. 78.

Parameters

G
[`networkx.Graph`] The connected graph G.

weight
[`str`, optional] Passed through to `shortest_path_length()`.

attr
[`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 `barycenter()` if `sp` is given but is missing shortest path lengths for any pairs.

ValueError
If `sp` and `weight` are both given.

See also:

`center`
`periphery`

Examples

```
>>> G = nx.Graph([(1, 2), (1, 3), (1, 4), (3, 4), (3, 5), (4, 5)])
>>> nx.barycenter(G)
[1, 3, 4]
```

3.23.2 center

center(*G*, *e=None*, *usebounds=False*, *weight=None*)

Returns the center of the graph G.

The center is the set of nodes with eccentricity equal to radius.

Parameters

G
[NetworkX graph] A graph

e
[eccentricity dictionary, optional] A precomputed dictionary of eccentricities.

weight
[string, function, or None] 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.

If this is None, every edge has weight/distance/cost 1.

Weights stored as floating point values can lead to small round-off errors in distances. Use integer weights to avoid this.

Weights should be positive, since they are distances.

Returns

c
[list] List of nodes in center

See also:

barycenter
periphery

Examples

```
>>> G = nx.Graph([(1, 2), (1, 3), (1, 4), (3, 4), (3, 5), (4, 5)])
>>> list(nx.center(G))
[1, 3, 4]
```

3.23.3 diameter

diameter(*G*, *e=None*, *usebounds=False*, *weight=None*)

Returns the diameter of the graph G.

The diameter is the maximum eccentricity.

Parameters

G
[NetworkX graph] A graph

e
: [eccentricity dictionary, optional] A precomputed dictionary of eccentricities.

weight
: [string, function, or None] 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.

  If this is None, every edge has weight/distance/cost 1.

  Weights stored as floating point values can lead to small round-off errors in distances. Use integer weights to avoid this.

  Weights should be positive, since they are distances.

Returns

d
: [integer] Diameter of graph

See also:

eccentricity

Examples

```
>>> G = nx.Graph([(1, 2), (1, 3), (1, 4), (3, 4), (3, 5), (4, 5)])
>>> nx.diameter(G)
3
```

3.23.4 eccentricity

eccentricity(*G*, *v=None*, *sp=None*, *weight=None*)

Returns the eccentricity of nodes in G.

The eccentricity of a node v is the maximum distance from v to all other nodes in G.

Parameters

G
: [NetworkX graph] A graph

v
: [node, optional] Return value of specified node

sp
: [dict of dicts, optional] All pairs shortest path lengths as a dictionary of dictionaries

weight
: [string, function, or None (default=None)] 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.

If this is None, every edge has weight/distance/cost 1.

Weights stored as floating point values can lead to small round-off errors in distances. Use integer weights to avoid this.

Weights should be positive, since they are distances.

Returns

ecc
[dictionary] A dictionary of eccentricity values keyed by node.

Examples

```
>>> G = nx.Graph([(1, 2), (1, 3), (1, 4), (3, 4), (3, 5), (4, 5)])
>>> dict(nx.eccentricity(G))
{1: 2, 2: 3, 3: 2, 4: 2, 5: 3}
```

```
>>> dict(nx.eccentricity(G, v=[1, 5]))  # This returns the eccentrity of node 1 &␣
↪5
{1: 2, 5: 3}
```

3.23.5 periphery

periphery(*G*, *e=None*, *usebounds=False*, *weight=None*)

Returns the periphery of the graph G.

The periphery is the set of nodes with eccentricity equal to the diameter.

Parameters

G
[NetworkX graph] A graph

e
[eccentricity dictionary, optional] A precomputed dictionary of eccentricities.

weight
[string, function, or None] 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.

If this is None, every edge has weight/distance/cost 1.

Weights stored as floating point values can lead to small round-off errors in distances. Use integer weights to avoid this.

Weights should be positive, since they are distances.

Returns

p
[list] List of nodes in periphery

See also:

barycenter
center

Examples

```
>>> G = nx.Graph([(1, 2), (1, 3), (1, 4), (3, 4), (3, 5), (4, 5)])
>>> nx.periphery(G)
[2, 5]
```

3.23.6 radius

radius(*G*, *e=None*, *usebounds=False*, *weight=None*)

Returns the radius of the graph G.

The radius is the minimum eccentricity.

Parameters

G
[NetworkX graph] A graph

e
[eccentricity dictionary, optional] A precomputed dictionary of eccentricities.

weight
[string, function, or None] 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.

If this is None, every edge has weight/distance/cost 1.

Weights stored as floating point values can lead to small round-off errors in distances. Use integer weights to avoid this.

Weights should be positive, since they are distances.

Returns

r
[integer] Radius of graph

Examples

```
>>> G = nx.Graph([(1, 2), (1, 3), (1, 4), (3, 4), (3, 5), (4, 5)])
>>> nx.radius(G)
2
```

3.23.7 resistance_distance

resistance_distance(*G*, *nodeA*, *nodeB*, *weight=None*, *invert_weight=True*)

Returns the resistance distance between node A and node B on graph G.

The resistance distance between two nodes of a graph is akin to treating the graph as a grid of resistorses with a resistance equal to the provided weight.

If weight is not provided, then a weight of 1 is used for all edges.

Parameters

G
[NetworkX graph] A graph

nodeA
[node] A node within graph G.

nodeB
[node] A node within graph G, exclusive of Node A.

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
[float] Value of effective resistance distance

Notes

Overviews are provided in [1] and [2]. Additional details on computational methods, proofs of properties, and corresponding MATLAB codes are provided in [3].

References

[1], [2], [3]

Examples

```
>>> G = nx.Graph([(1, 2), (1, 3), (1, 4), (3, 4), (3, 5), (4, 5)])
>>> nx.resistance_distance(G, 1, 3)
0.625
```

3.24 Distance-Regular Graphs

is_distance_regular(G)	Returns True if the graph is distance regular, False otherwise.
is_strongly_regular(G)	Returns True if and only if the given graph is strongly regular.
intersection_array(G)	Returns the intersection array of a distance-regular graph.
global_parameters(b, c)	Returns global parameters for a given intersection array.

3.24.1 is_distance_regular

is_distance_regular(*G*)

Returns True if the graph is distance regular, False otherwise.

A connected graph G is distance-regular if for any nodes x,y and any integers i,j=0,1,…,d (where d is the graph diameter), the number of vertices at distance i from x and distance j from y depends only on i,j and the graph distance between x and y, independently of the choice of x and y.

Parameters

G: Networkx graph (undirected)

Returns

bool
True if the graph is Distance Regular, False otherwise

See also:

intersection_array, ***global_parameters***

Notes

For undirected and simple graphs only

References

[1], [2]

Examples

```
>>> G = nx.hypercube_graph(6)
>>> nx.is_distance_regular(G)
True
```

3.24.2 is_strongly_regular

is_strongly_regular(*G*)

Returns True if and only if the given graph is strongly regular.

An undirected graph is *strongly regular* if

- it is regular,
- each pair of adjacent vertices has the same number of neighbors in common,
- each pair of nonadjacent vertices has the same number of neighbors in common.

Each strongly regular graph is a distance-regular graph. Conversely, if a distance-regular graph has diameter two, then it is a strongly regular graph. For more information on distance-regular graphs, see *is_distance_regular()*.

Parameters

G
[NetworkX graph] An undirected graph.

Returns

bool
Whether G is strongly regular.

Examples

The cycle graph on five vertices is strongly regular. It is two-regular, each pair of adjacent vertices has no shared neighbors, and each pair of nonadjacent vertices has one shared neighbor:

```
>>> G = nx.cycle_graph(5)
>>> nx.is_strongly_regular(G)
True
```

3.24.3 intersection_array

intersection_array(*G*)

Returns the intersection array of a distance-regular graph.

Given a distance-regular graph G with integers b_i, c_i,i = 0,….,d such that for any 2 vertices x,y in G at a distance i=d(x,y), there are exactly c_i neighbors of y at a distance of i-1 from x and b_i neighbors of y at a distance of i+1 from x.

A distance regular graph's intersection array is given by, [b_0,b_1,…..b_{d-1};c_1,c_2,…..c_d]

Parameters

G: Networkx graph (undirected)

Returns

b,c: tuple of lists

See also:

global_parameters

References

[1]

Examples

```
>>> G = nx.icosahedral_graph()
>>> nx.intersection_array(G)
([5, 2, 1], [1, 2, 5])
```

3.24.4 global_parameters

global_parameters(*b*, *c*)

Returns global parameters for a given intersection array.

Given a distance-regular graph G with integers b_i, c_i,i = 0,….,d such that for any 2 vertices x,y in G at a distance i=d(x,y), there are exactly c_i neighbors of y at a distance of i-1 from x and b_i neighbors of y at a distance of i+1 from x.

Thus, a distance regular graph has the global parameters, [[c_0,a_0,b_0],[c_1,a_1,b_1],…….,[c_d,a_d,b_d]] for the intersection array [b_0,b_1,…..b_{d-1};c_1,c_2,…..c_d] where a_i+b_i+c_i=k , k= degree of every vertex.

Parameters

b
[list]

c
[list]

Returns

iterable
An iterable over three tuples.

See also:

intersection_array

References

[1]

Examples

```
>>> G = nx.dodecahedral_graph()
>>> b, c = nx.intersection_array(G)
>>> list(nx.global_parameters(b, c))
[(0, 0, 3), (1, 0, 2), (1, 1, 1), (1, 1, 1), (2, 0, 1), (3, 0, 0)]
```

3.25 Dominance

Dominance algorithms.

immediate_dominators(G, start)	Returns the immediate dominators of all nodes of a directed graph.
dominance_frontiers(G, start)	Returns the dominance frontiers of all nodes of a directed graph.

3.25.1 immediate_dominators

immediate_dominators(*G*, *start*)

Returns the immediate dominators 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

idom
[dict keyed by nodes] A dict containing the immediate dominators of each node reachable from `start`.

Raises

NetworkXNotImplemented
If `G` is undirected.

NetworkXError
If `start` is not in `G`.

Notes

Except for `start`, the immediate dominators are the parents of their corresponding nodes in the dominator tree.

References

[1]

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)]
```

3.25.2 dominance_frontiers

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`.

References

[1]

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, [])]
```

3.26 Dominating Sets

Functions for computing dominating sets in a graph.

dominating_set(G[, start_with])	Finds a dominating set for the graph G.
is_dominating_set(G, nbunch)	Checks if `nbunch` is a dominating set for `G`.

3.26.1 dominating_set

dominating_set(*G*, *start_with=None*)

Finds a dominating set for the graph G.

A *dominating set* for a graph with node set V is a subset D of V such that every node not in D is adjacent to at least one member of D [1].

Parameters

G
[NetworkX graph]

start_with
[node (default=None)] Node to use as a starting point for the algorithm.

Returns

D
[set] A dominating set for G.

See also:

is_dominating_set

Notes

This function is an implementation of algorithm 7 in [2] which finds some dominating set, not necessarily the smallest one.

References

[1], [2]

3.26.2 is_dominating_set

is_dominating_set(*G*, *nbunch*)

Checks if `nbunch` is a dominating set for `G`.

A *dominating set* for a graph with node set V is a subset D of V such that every node not in D is adjacent to at least one member of D [1].

Parameters

G
[NetworkX graph]

nbunch
[iterable] An iterable of nodes in the graph `G`.

See also:

dominating_set

References

[1]

3.27 Efficiency

Provides functions for computing the efficiency of nodes and graphs.

efficiency(G, u, v)	Returns the efficiency of a pair of nodes in a graph.
local_efficiency(G)	Returns the average local efficiency of the graph.
global_efficiency(G)	Returns the average global efficiency of the graph.

3.27.1 efficiency

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
[*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.

See also:

local_efficiency
global_efficiency

Notes

Edge weights are ignored when computing the shortest path distances.

References

[1]

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
```

3.27.2 local_efficiency

local_efficiency(*G*)

Returns the average local 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 *local efficiency* of a node in the graph is the average global efficiency of the subgraph induced by the neighbors of the node. The *average local efficiency* is the average of the local efficiencies of each node [1].

Parameters

G
[*networkx.Graph*] An undirected graph for which to compute the average local efficiency.

Returns

float
The average local efficiency of the graph.

See also:

global_efficiency

Notes

Edge weights are ignored when computing the shortest path distances.

References

[1]

Examples

```
>>> G = nx.Graph([(0, 1), (0, 2), (0, 3), (1, 2), (1, 3)])
>>> nx.local_efficiency(G)
0.9166666666666667
```

3.27.3 global_efficiency

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
[*networkx.Graph*] An undirected graph for which to compute the average global efficiency.

Returns

float
The average global efficiency of the graph.

See also:

local_efficiency

Notes

Edge weights are ignored when computing the shortest path distances.

References

[1]

Examples

```
>>> G = nx.Graph([(0, 1), (0, 2), (0, 3), (1, 2), (1, 3)])
>>> round(nx.global_efficiency(G), 12)
0.916666666667
```

3.28 Eulerian

Eulerian circuits and graphs.

is_eulerian(G)	Returns True if and only if G is Eulerian.
eulerian_circuit(G[, source, keys])	Returns an iterator over the edges of an Eulerian circuit in G.
eulerize(G)	Transforms a graph into an Eulerian graph.
is_semieulerian(G)	Return True iff G is semi-Eulerian.
has_eulerian_path(G[, source])	Return True iff G has an Eulerian path.
eulerian_path(G[, source, keys])	Return an iterator over the edges of an Eulerian path in G.

3.28.1 is_eulerian

is_eulerian(*G*)

Returns True if and only if G is Eulerian.

A graph is *Eulerian* if it has an Eulerian circuit. An *Eulerian circuit* is a closed walk that includes each edge of a graph exactly once.

Graphs with isolated vertices (i.e. vertices with zero degree) are not considered to have Eulerian circuits. Therefore, if the graph is not connected (or not strongly connected, for directed graphs), this function returns False.

Parameters

G

[NetworkX graph] A graph, either directed or undirected.

Examples

```
>>> nx.is_eulerian(nx.DiGraph({0: [3], 1: [2], 2: [3], 3: [0, 1]}))
True
>>> nx.is_eulerian(nx.complete_graph(5))
True
>>> nx.is_eulerian(nx.petersen_graph())
False
```

If you prefer to allow graphs with isolated vertices to have Eulerian circuits, you can first remove such vertices and then call *is_eulerian* as below example shows.

```
>>> G = nx.Graph([(0, 1), (1, 2), (0, 2)])
>>> G.add_node(3)
>>> nx.is_eulerian(G)
False
```

```
>>> G.remove_nodes_from(list(nx.isolates(G)))
>>> nx.is_eulerian(G)
True
```

3.28.2 eulerian_circuit

eulerian_circuit(*G*, *source=None*, *keys=False*)

Returns an iterator over the edges of an Eulerian circuit in `G`.

An *Eulerian circuit* is a closed walk that includes each edge of a graph exactly once.

Parameters

G
[NetworkX graph] A graph, either directed or undirected.

source
[node, optional] Starting node for circuit.

keys
[bool] If False, edges generated by this function will be of the form `(u, v)`. Otherwise, edges will be of the form `(u, v, k)`. This option is ignored unless `G` is a multigraph.

Returns

edges
[iterator] An iterator over edges in the Eulerian circuit.

Raises

NetworkXError
If the graph is not Eulerian.

See also:

is_eulerian

Notes

This is a linear time implementation of an algorithm adapted from [1].

For general information about Euler tours, see [2].

References

[1], [2]

Examples

To get an Eulerian circuit in an undirected graph:

```
>>> G = nx.complete_graph(3)
>>> list(nx.eulerian_circuit(G))
[(0, 2), (2, 1), (1, 0)]
>>> list(nx.eulerian_circuit(G, source=1))
[(1, 2), (2, 0), (0, 1)]
```

To get the sequence of vertices in an Eulerian circuit:

```
>>> [u for u, v in nx.eulerian_circuit(G)]
[0, 2, 1]
```

3.28.3 eulerize

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], [2], [3]

Examples

```
>>> G = nx.complete_graph(10)
>>> H = nx.eulerize(G)
>>> nx.is_eulerian(H)
True
```

3.28.4 is_semieulerian

is_semieulerian(*G*)

Return True iff `G` is semi-Eulerian.

G is semi-Eulerian if it has an Eulerian path but no Eulerian circuit.

See also:

has_eulerian_path
is_eulerian

3.28.5 has_eulerian_path

has_eulerian_path(*G*, *source=None*)

Return True iff `G` has an Eulerian path.

An Eulerian path is a path in a graph which uses each edge of a graph exactly once. If `source` is specified, then this function checks whether an Eulerian path that starts at node `source` exists.

A directed graph has an Eulerian path iff:

- at most one vertex has out_degree - in_degree = 1,
- at most one vertex has in_degree - out_degree = 1,
- every other vertex has equal in_degree and out_degree,
- and all of its vertices belong to a single connected component of the underlying undirected graph.

If `source` is not None, an Eulerian path starting at `source` exists if no other node has out_degree - in_degree = 1. This is equivalent to either there exists an Eulerian circuit or `source` has out_degree - in_degree = 1 and the conditions above hold.

An undirected graph has an Eulerian path iff:

- exactly zero or two vertices have odd degree,
- and all of its vertices belong to a single connected component.

If `source` is not None, an Eulerian path starting at `source` exists if either there exists an Eulerian circuit or `source` has an odd degree and the conditions above hold.

Graphs with isolated vertices (i.e. vertices with zero degree) are not considered to have an Eulerian path. Therefore, if the graph is not connected (or not strongly connected, for directed graphs), this function returns False.

Parameters

G
[NetworkX Graph] The graph to find an euler path in.

source
[node, optional] Starting node for path.

Returns

Bool
[True if G has an Eulerian path.]

See also:

is_eulerian
eulerian_path

Examples

If you prefer to allow graphs with isolated vertices to have Eulerian path, you can first remove such vertices and then call *has_eulerian_path* as below example shows.

```
>>> G = nx.Graph([(0, 1), (1, 2), (0, 2)])
>>> G.add_node(3)
>>> nx.has_eulerian_path(G)
False
```

```
>>> G.remove_nodes_from(list(nx.isolates(G)))
>>> nx.has_eulerian_path(G)
True
```

3.28.6 eulerian_path

eulerian_path(*G*, *source=None*, *keys=False*)

Return an iterator over the edges of an Eulerian path in G.

Parameters

G
[NetworkX Graph] The graph in which to look for an eulerian path.

source
[node or None (default: None)] The node at which to start the search. None means search over all 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.

3.29 Flows

3.29.1 Maximum Flow

maximum_flow(flowG, _s, _t[, capacity, ...])	Find a maximum single-commodity flow.
maximum_flow_value(flowG, _s, _t[, ...])	Find the value of maximum single-commodity flow.
minimum_cut(flowG, _s, _t[, capacity, flow_func])	Compute the value and the node partition of a minimum (s, t)-cut.
minimum_cut_value(flowG, _s, _t[, capacity, ...])	Compute the value of a minimum (s, t)-cut.

maximum_flow

maximum_flow(*flowG*, *_s*, *_t*, *capacity='capacity'*, *flow_func=None*, ***kwargs*)

Find a maximum single-commodity flow.

Parameters

flowG
[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'.

flow_func
[function] A function for computing the maximum flow among a pair of nodes in a capacitated graph. The function has to accept at least three parameters: a Graph or Digraph, a source node, and a target node. And return a residual network that follows NetworkX conventions (see Notes). If flow_func is None, the default maximum flow function (*preflow_push()*) is used. See below for alternative algorithms. The choice of the default function may change from version to version and should not be relied on. Default value: None.

kwargs
[Any other keyword parameter is passed to the function that] computes the maximum flow.

Returns

flow_value
[integer, float] Value of the maximum flow, i.e., net outflow from the source.

flow_dict
[dict] A dictionary containing the value of the flow that went through each edge.

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:

maximum_flow_value()
minimum_cut()
minimum_cut_value()
edmonds_karp()
preflow_push()
shortest_augmenting_path()

Notes

The function used in the flow_func parameter has to return a residual network that follows NetworkX conventions:

The residual network `R` from an input graph `G` has the same nodes as `G`. `R` is a DiGraph that contains a pair of edges `(u, v)` and `(v, u)` iff `(u, v)` is not a self-loop, and at least one of `(u, v)` and `(v, u)` exists in `G`.

For each edge `(u, v)` in `R`, `R[u][v]['capacity']` is equal to the capacity of `(u, v)` in `G` if it exists in `G` or zero otherwise. If the capacity is infinite, `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 `R.graph['inf']`. For each edge `(u, v)` in `R`, `R[u][v]['flow']` represents the flow function of `(u, v)` and satisfies `R[u][v]['flow'] == -R[v][u]['flow']`.

The flow value, defined as the total flow into `t`, the sink, is stored in `R.graph['flow_value']`. Reachability to `t` using only edges `(u, v)` such that `R[u][v]['flow'] < R[u][v]['capacity']` induces a minimum `s-t` cut.

Specific algorithms may store extra data in `R`.

The function should supports an optional boolean parameter value_only. When True, it can optionally terminate the algorithm as soon as the maximum flow value and the minimum cut can be determined.

Examples

```
>>> 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)
```

maximum_flow returns both the value of the maximum flow and a dictionary with all flows.

```
>>> flow_value, flow_dict = nx.maximum_flow(G, "x", "y")
>>> flow_value
3.0
>>> print(flow_dict["x"]["b"])
1.0
```

You can also use alternative algorithms for computing the maximum flow by using the flow_func parameter.

```
>>> from networkx.algorithms.flow import shortest_augmenting_path
>>> flow_value == nx.maximum_flow(G, "x", "y", flow_func=shortest_augmenting_
↪path)[
...     0
... ]
True
```

maximum_flow_value

maximum_flow_value(*flowG*, *_s*, *_t*, *capacity='capacity'*, *flow_func=None*, ***kwargs*)

Find the value of maximum single-commodity flow.

Parameters

flowG
[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'.

flow_func
[function] A function for computing the maximum flow among a pair of nodes in a capacitated graph. The function has to accept at least three parameters: a Graph or Digraph, a source node, and a target node. And return a residual network that follows NetworkX conventions (see Notes). If flow_func is None, the default maximum flow function (*preflow_push()*) is used. See below for alternative algorithms. The choice of the default function may change from version to version and should not be relied on. Default value: None.

kwargs
[Any other keyword parameter is passed to the function that] computes the maximum flow.

Returns

flow_value
[integer, float] Value of the maximum flow, i.e., net outflow from the source.

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:

maximum_flow()
minimum_cut()
minimum_cut_value()
edmonds_karp()
preflow_push()
shortest_augmenting_path()

Notes

The function used in the flow_func parameter has to return a residual network that follows NetworkX conventions:

The residual network `R` from an input graph `G` has the same nodes as `G`. `R` is a DiGraph that contains a pair of edges `(u, v)` and `(v, u)` iff `(u, v)` is not a self-loop, and at least one of `(u, v)` and `(v, u)` exists in `G`.

For each edge `(u, v)` in `R`, `R[u][v]['capacity']` is equal to the capacity of `(u, v)` in `G` if it exists in `G` or zero otherwise. If the capacity is infinite, `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 `R.graph['inf']`. For each edge `(u, v)` in `R`, `R[u][v]['flow']` represents the flow function of `(u, v)` and satisfies `R[u][v]['flow'] == -R[v][u]['flow']`.

The flow value, defined as the total flow into `t`, the sink, is stored in `R.graph['flow_value']`. Reachability to `t` using only edges `(u, v)` such that `R[u][v]['flow'] < R[u][v]['capacity']` induces a minimum `s-t` cut.

Specific algorithms may store extra data in `R`.

The function should supports an optional boolean parameter value_only. When True, it can optionally terminate the algorithm as soon as the maximum flow value and the minimum cut can be determined.

Examples

```
>>> 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)
```

maximum_flow_value computes only the value of the maximum flow:

```
>>> flow_value = nx.maximum_flow_value(G, "x", "y")
>>> flow_value
3.0
```

You can also use alternative algorithms for computing the maximum flow by using the flow_func parameter.

```
>>> from networkx.algorithms.flow import shortest_augmenting_path
>>> flow_value == nx.maximum_flow_value(
...     G, "x", "y", flow_func=shortest_augmenting_path
... )
True
```

minimum_cut

minimum_cut(*flowG*, *_s*, *_t*, *capacity='capacity'*, *flow_func=None*, ***kwargs*)

Compute the value and the node partition of a minimum (s, t)-cut.

Use the max-flow min-cut theorem, i.e., the capacity of a minimum capacity cut is equal to the flow value of a maximum flow.

Parameters

flowG
[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'.

flow_func
[function] A function for computing the maximum flow among a pair of nodes in a capacitated graph. The function has to accept at least three parameters: a Graph or Digraph, a source node, and a target node. And return a residual network that follows NetworkX conventions (see Notes). If flow_func is None, the default maximum flow function (*preflow_push()*) is used. See below for alternative algorithms. The choice of the default function may change from version to version and should not be relied on. Default value: None.

kwargs
[Any other keyword parameter is passed to the function that] computes the maximum flow.

Returns

cut_value
[integer, float] Value of the minimum cut.

partition
[pair of node sets] A partitioning of the nodes that defines a minimum cut.

Raises

NetworkXUnbounded
If the graph has a path of infinite capacity, all cuts have infinite capacity and the function raises a NetworkXError.

See also:

maximum_flow()
maximum_flow_value()
minimum_cut_value()
edmonds_karp()
preflow_push()
shortest_augmenting_path()

Notes

The function used in the flow_func parameter has to return a residual network that follows NetworkX conventions:

The residual network `R` from an input graph `G` has the same nodes as `G`. `R` is a DiGraph that contains a pair of edges `(u, v)` and `(v, u)` iff `(u, v)` is not a self-loop, and at least one of `(u, v)` and `(v, u)` exists in `G`.

For each edge `(u, v)` in `R`, `R[u][v]['capacity']` is equal to the capacity of `(u, v)` in `G` if it exists in `G` or zero otherwise. If the capacity is infinite, `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 `R.graph['inf']`. For each edge `(u, v)` in `R`, `R[u][v]['flow']` represents the flow function of `(u, v)` and satisfies `R[u][v]['flow'] == -R[v][u]['flow']`.

The flow value, defined as the total flow into `t`, the sink, is stored in `R.graph['flow_value']`. Reachability to `t` using only edges `(u, v)` such that `R[u][v]['flow'] < R[u][v]['capacity']` induces a minimum `s-t` cut.

Specific algorithms may store extra data in `R`.

The function should supports an optional boolean parameter value_only. When True, it can optionally terminate the algorithm as soon as the maximum flow value and the minimum cut can be determined.

Examples

```
>>> 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)
```

minimum_cut computes both the value of the minimum cut and the node partition:

```
>>> cut_value, partition = nx.minimum_cut(G, "x", "y")
>>> reachable, non_reachable = partition
```

'partition' here is a tuple with the two sets of nodes that define the minimum cut. You can compute the cut set of edges that induce the minimum cut as follows:

```
>>> cutset = set()
>>> for u, nbrs in ((n, G[n]) for n in reachable):
...     cutset.update((u, v) for v in nbrs if v in non_reachable)
>>> print(sorted(cutset))
[('c', 'y'), ('x', 'b')]
>>> cut_value == sum(G.edges[u, v]["capacity"] for (u, v) in cutset)
True
```

You can also use alternative algorithms for computing the minimum cut by using the flow_func parameter.

```
>>> from networkx.algorithms.flow import shortest_augmenting_path
>>> cut_value == nx.minimum_cut(G, "x", "y", flow_func=shortest_augmenting_
↪path)[0]
True
```

minimum_cut_value

minimum_cut_value(*flowG*, *_s*, *_t*, *capacity='capacity'*, *flow_func=None*, ***kwargs*)

Compute the value of a minimum (s, t)-cut.

Use the max-flow min-cut theorem, i.e., the capacity of a minimum capacity cut is equal to the flow value of a maximum flow.

Parameters

flowG
[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'.

flow_func
[function] A function for computing the maximum flow among a pair of nodes in a capacitated graph. The function has to accept at least three parameters: a Graph or Digraph, a source node, and a target node. And return a residual network that follows NetworkX conventions (see Notes). If flow_func is None, the default maximum flow function (*preflow_push()*) is used. See below for alternative algorithms. The choice of the default function may change from version to version and should not be relied on. Default value: None.

kwargs
[Any other keyword parameter is passed to the function that] computes the maximum flow.

Returns

cut_value
[integer, float] Value of the minimum cut.

Raises

NetworkXUnbounded
If the graph has a path of infinite capacity, all cuts have infinite capacity and the function raises a NetworkXError.

See also:

maximum_flow()
maximum_flow_value()
minimum_cut()
edmonds_karp()
preflow_push()
shortest_augmenting_path()

Notes

The function used in the flow_func parameter has to return a residual network that follows NetworkX conventions:

The residual network `R` from an input graph `G` has the same nodes as `G`. `R` is a DiGraph that contains a pair of edges `(u, v)` and `(v, u)` iff `(u, v)` is not a self-loop, and at least one of `(u, v)` and `(v, u)` exists in `G`.

For each edge `(u, v)` in `R`, `R[u][v]['capacity']` is equal to the capacity of `(u, v)` in `G` if it exists in `G` or zero otherwise. If the capacity is infinite, `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 `R.graph['inf']`. For each edge `(u, v)` in `R`, `R[u][v]['flow']` represents the flow function of `(u, v)` and satisfies `R[u][v]['flow'] == -R[v][u]['flow']`.

The flow value, defined as the total flow into `t`, the sink, is stored in `R.graph['flow_value']`. Reachability to `t` using only edges `(u, v)` such that `R[u][v]['flow'] < R[u][v]['capacity']` induces a minimum `s-t` cut.

Specific algorithms may store extra data in `R`.

The function should supports an optional boolean parameter value_only. When True, it can optionally terminate the algorithm as soon as the maximum flow value and the minimum cut can be determined.

Examples

```
>>> 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)
```

minimum_cut_value computes only the value of the minimum cut:

```
>>> cut_value = nx.minimum_cut_value(G, "x", "y")
>>> cut_value
3.0
```

You can also use alternative algorithms for computing the minimum cut by using the flow_func parameter.

```
>>> from networkx.algorithms.flow import shortest_augmenting_path
>>> cut_value == nx.minimum_cut_value(
...     G, "x", "y", flow_func=shortest_augmenting_path
... )
True
```

3.29.2 Edmonds-Karp

edmonds_karp(G, s, t[, capacity, residual, ...])	Find a maximum single-commodity flow using the Edmonds-Karp algorithm.

edmonds_karp

edmonds_karp(*G*, *s*, *t*, *capacity='capacity'*, *residual=None*, *value_only=False*, *cutoff=None*)

Find a maximum single-commodity flow using the Edmonds-Karp 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 $O(nm^2)$ for n nodes and 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.

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:

maximum_flow()
minimum_cut()
preflow_push()
shortest_augmenting_path()

Notes

The residual network `R` from an input graph `G` has the same nodes as `G`. `R` is a DiGraph that contains a pair of edges `(u, v)` and `(v, u)` iff `(u, v)` is not a self-loop, and at least one of `(u, v)` and `(v, u)` exists in `G`.

For each edge `(u, v)` in `R`, `R[u][v]['capacity']` is equal to the capacity of `(u, v)` in `G` if it exists in `G` or zero otherwise. If the capacity is infinite, `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 `R.graph['inf']`. For each edge `(u, v)` in `R`, `R[u][v]['flow']` represents the flow function of `(u, v)` and satisfies `R[u][v]['flow'] == -R[v][u]['flow']`.

The flow value, defined as the total flow into `t`, the sink, is stored in `R.graph['flow_value']`. If `cutoff` is not specified, reachability to `t` using only edges `(u, v)` such that `R[u][v]['flow'] < R[u][v]['capacity']` induces a minimum `s-t` cut.

Examples

```
>>> from networkx.algorithms.flow import edmonds_karp
```

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 = edmonds_karp(G, "x", "y")
>>> flow_value = nx.maximum_flow_value(G, "x", "y")
>>> flow_value
3.0
>>> flow_value == R.graph["flow_value"]
True
```

3.29.3 Shortest Augmenting Path

shortest_augmenting_path(G, s, t[, ...])	Find a maximum single-commodity flow using the shortest augmenting path algorithm.

shortest_augmenting_path

shortest_augmenting_path(*G*, *s*, *t*, *capacity='capacity'*, *residual=None*, *value_only=False*, *two_phase=False*, *cutoff=None*)

Find a maximum single-commodity flow using the shortest augmenting path 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.

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.

two_phase
[bool] If True, a two-phase variant is used. The two-phase variant improves the running time on unit-capacity networks from $O(nm)$ to $O(\min(n^{2/3}, m^{1/2})m)$. 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:

maximum_flow()
minimum_cut()
edmonds_karp()
preflow_push()

Notes

The residual network `R` from an input graph `G` has the same nodes as `G`. `R` is a DiGraph that contains a pair of edges `(u, v)` and `(v, u)` iff `(u, v)` is not a self-loop, and at least one of `(u, v)` and `(v, u)` exists in `G`.

For each edge `(u, v)` in `R`, `R[u][v]['capacity']` is equal to the capacity of `(u, v)` in `G` if it exists in `G` or zero otherwise. If the capacity is infinite, `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 `R.graph['inf']`. For each edge `(u, v)` in `R`, `R[u][v]['flow']` represents the flow function of `(u, v)` and satisfies `R[u][v]['flow'] == -R[v][u]['flow']`.

The flow value, defined as the total flow into `t`, the sink, is stored in `R.graph['flow_value']`. If `cutoff` is not specified, reachability to `t` using only edges `(u, v)` such that `R[u][v]['flow'] < R[u][v]['capacity']` induces a minimum `s-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
```

3.29.4 Preflow-Push

preflow_push(G, s, t[, capacity, residual, ...])	Find a maximum single-commodity flow using the highest-label preflow-push algorithm.

preflow_push

preflow_push(*G*, *s*, *t*, *capacity='capacity'*, *residual=None*, *global_relabel_freq=1*, *value_only=False*)

Find a maximum single-commodity flow using the highest-label preflow-push 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 $O(n^2\sqrt{m})$ for n nodes and 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:

maximum_flow()
minimum_cut()
edmonds_karp()
shortest_augmenting_path()

Notes

The residual network `R` from an input graph `G` has the same nodes as `G`. `R` is a DiGraph that contains a pair of edges `(u, v)` and `(v, u)` iff `(u, v)` is not a self-loop, and at least one of `(u, v)` and `(v, u)` exists in `G`. For each node `u` in `R`, `R.nodes[u]['excess']` represents the difference between flow into `u` and flow out of `u`.

For each edge `(u, v)` in `R`, `R[u][v]['capacity']` is equal to the capacity of `(u, v)` in `G` if it exists in `G` or zero otherwise. If the capacity is infinite, `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 `R.graph['inf']`. For each edge `(u, v)` in `R`, `R[u][v]['flow']` represents the flow function of `(u, v)` and satisfies `R[u][v]['flow'] == -R[v][u]['flow']`.

The flow value, defined as the total flow into `t`, the sink, is stored in `R.graph['flow_value']`. Reachability to `t` using only edges `(u, v)` such that `R[u][v]['flow'] < R[u][v]['capacity']` induces a minimum `s-t` cut.

Examples

```
>>> from networkx.algorithms.flow import preflow_push
```

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 = preflow_push(G, "x", "y")
>>> flow_value = nx.maximum_flow_value(G, "x", "y")
>>> flow_value == R.graph["flow_value"]
True
>>> # preflow_push also stores the maximum flow value
>>> # in the excess attribute of the sink node t
>>> flow_value == R.nodes["y"]["excess"]
True
>>> # For some problems, you might only want to compute a
>>> # maximum preflow.
>>> R = preflow_push(G, "x", "y", value_only=True)
>>> flow_value == R.graph["flow_value"]
True
>>> flow_value == R.nodes["y"]["excess"]
True
```

3.29.5 Dinitz

dinitz(G, s, t[, capacity, residual, ...])	Find a maximum single-commodity flow using Dinitz' algorithm.

dinitz

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:

maximum_flow()
minimum_cut()
preflow_push()
shortest_augmenting_path()

Notes

The residual network `R` from an input graph `G` has the same nodes as `G`. `R` is a DiGraph that contains a pair of edges `(u, v)` and `(v, u)` iff `(u, v)` is not a self-loop, and at least one of `(u, v)` and `(v, u)` exists in `G`.

For each edge `(u, v)` in `R`, `R[u][v]['capacity']` is equal to the capacity of `(u, v)` in `G` if it exists in `G` or zero otherwise. If the capacity is infinite, `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 `R.graph['inf']`. For each edge `(u, v)` in `R`, `R[u][v]['flow']` represents the flow function of `(u, v)` and satisfies `R[u][v]['flow'] == -R[v][u]['flow']`.

The flow value, defined as the total flow into `t`, the sink, is stored in `R.graph['flow_value']`. If `cutoff` is not specified, reachability to `t` using only edges `(u, v)` such that `R[u][v]['flow'] < R[u][v]['capacity']` induces a minimum `s-t` cut.

References

[1]

Examples

```
>>> from networkx.algorithms.flow import dinitz
```

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 = dinitz(G, "x", "y")
>>> flow_value = nx.maximum_flow_value(G, "x", "y")
>>> flow_value
3.0
>>> flow_value == R.graph["flow_value"]
True
```

3.29.6 Boykov-Kolmogorov

boykov_kolmogorov(G, s, t[, capacity, ...])	Find a maximum single-commodity flow using Boykov-Kolmogorov algorithm.

boykov_kolmogorov

boykov_kolmogorov(*G*, *s*, *t*, *capacity='capacity'*, *residual=None*, *value_only=False*, *cutoff=None*)

Find a maximum single-commodity flow using Boykov-Kolmogorov 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 worse case complexity $O(n^2m|C|)$ for n nodes, m edges, and $|C|$ the cost of the minimum cut [1]. This implementation uses the marking heuristic defined in [2] which improves its running time in many practical problems.

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:

maximum_flow()
minimum_cut()
preflow_push()
shortest_augmenting_path()

Notes

The residual network `R` from an input graph `G` has the same nodes as `G`. `R` is a DiGraph that contains a pair of edges `(u, v)` and `(v, u)` iff `(u, v)` is not a self-loop, and at least one of `(u, v)` and `(v, u)` exists in `G`.

For each edge `(u, v)` in `R`, `R[u][v]['capacity']` is equal to the capacity of `(u, v)` in `G` if it exists in `G` or zero otherwise. If the capacity is infinite, `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 `R.graph['inf']`. For each edge `(u, v)` in `R`, `R[u][v]['flow']` represents the flow function of `(u, v)` and satisfies `R[u][v]['flow'] == -R[v][u]['flow']`.

The flow value, defined as the total flow into `t`, the sink, is stored in `R.graph['flow_value']`. If `cutoff` is not specified, reachability to `t` using only edges `(u, v)` such that `R[u][v]['flow'] < R[u][v]['capacity']` induces a minimum `s-t` cut.

References

[1], [2]

Examples

```
>>> from networkx.algorithms.flow import boykov_kolmogorov
```

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 = boykov_kolmogorov(G, "x", "y")
>>> flow_value = nx.maximum_flow_value(G, "x", "y")
>>> flow_value
3.0
>>> flow_value == R.graph["flow_value"]
True
```

A nice feature of the Boykov-Kolmogorov algorithm is that a partition of the nodes that defines a minimum cut can be easily computed based on the search trees used during the algorithm. These trees are stored in the graph attribute `trees` of the residual network.

```
>>> source_tree, target_tree = R.graph["trees"]
>>> partition = (set(source_tree), set(G) - set(source_tree))
```

Or equivalently:

```
>>> partition = (set(G) - set(target_tree), set(target_tree))
```

3.29.7 Gomory-Hu Tree

gomory_hu_tree(G[, capacity, flow_func])	Returns the Gomory-Hu tree of an undirected graph G.

gomory_hu_tree

gomory_hu_tree(*G*, *capacity='capacity'*, *flow_func=None*)

Returns the Gomory-Hu tree of an undirected graph G.

A Gomory-Hu tree of an undirected graph with capacities is a weighted tree that represents the minimum s-t cuts for all s-t pairs in the graph.

It only requires `n-1` minimum cut computations instead of the obvious `n(n-1)/2`. The tree represents all s-t cuts as the minimum cut value among any pair of nodes is the minimum edge weight in the shortest path between the two nodes in the Gomory-Hu tree.

The Gomory-Hu tree also has the property that removing the edge with the minimum weight in the shortest path between any two nodes leaves two connected components that form a partition of the nodes in G that defines the minimum s-t cut.

See Examples section below for details.

Parameters

G
[NetworkX graph] Undirected graph

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'.

flow_func
[function] Function to perform the underlying flow computations. Default value *edmonds_karp()*. This function performs better in sparse graphs with right tailed degree distributions. *shortest_augmenting_path()* will perform better in denser graphs.

Returns

Tree
[NetworkX graph] A NetworkX graph representing the Gomory-Hu tree of the input graph.

Raises

NetworkXNotImplemented
Raised if the input graph is directed.

NetworkXError
Raised if the input graph is an empty Graph.

See also:

minimum_cut()
maximum_flow()

Notes

This implementation is based on Gusfield approach [1] to compute Comory-Hu trees, which does not require node contractions and has the same computational complexity than the original method.

References

[1]

Examples

```
>>> G = nx.karate_club_graph()
>>> nx.set_edge_attributes(G, 1, "capacity")
>>> T = nx.gomory_hu_tree(G)
>>> # The value of the minimum cut between any pair
... # of nodes in G is the minimum edge weight in the
... # shortest path between the two nodes in the
... # Gomory-Hu tree.
... def minimum_edge_weight_in_shortest_path(T, u, v):
...     path = nx.shortest_path(T, u, v, weight="weight")
...     return min((T[u][v]["weight"], (u, v)) for (u, v) in zip(path, path[1:]))
>>> u, v = 0, 33
>>> cut_value, edge = minimum_edge_weight_in_shortest_path(T, u, v)
>>> cut_value
10
>>> nx.minimum_cut_value(G, u, v)
10
>>> # The Comory-Hu tree also has the property that removing the
... # edge with the minimum weight in the shortest path between
... # any two nodes leaves two connected components that form
... # a partition of the nodes in G that defines the minimum s-t
... # cut.
... cut_value, edge = minimum_edge_weight_in_shortest_path(T, u, v)
>>> T.remove_edge(*edge)
>>> U, V = list(nx.connected_components(T))
>>> # Thus U and V form a partition that defines a minimum cut
... # between u and v in G. You can compute the edge cut set,
... # that is, the set of edges that if removed from G will
... # disconnect u from v in G, with this information:
... cutset = set()
>>> for x, nbrs in ((n, G[n]) for n in U):
...     cutset.update((x, y) for y in nbrs if y in V)
>>> # Because we have set the capacities of all edges to 1
... # the cutset contains ten edges
... len(cutset)
10
```

(continues on next page)

(continued from previous page)

```
>>> # You can use any maximum flow algorithm for the underlying
... # flow computations using the argument flow_func
... from networkx.algorithms import flow
>>> T = nx.gomory_hu_tree(G, flow_func=flow.boykov_kolmogorov)
>>> cut_value, edge = minimum_edge_weight_in_shortest_path(T, u, v)
>>> cut_value
10
>>> nx.minimum_cut_value(G, u, v, flow_func=flow.boykov_kolmogorov)
10
```

3.29.8 Utils

build_residual_network(G, capacity)	Build a residual network and initialize a zero flow.

build_residual_network

build_residual_network (*G*, *capacity*)

Build a residual network and initialize a zero flow.

The residual network `R` from an input graph `G` has the same nodes as `G`. `R` is a DiGraph that contains a pair of edges `(u, v)` and `(v, u)` iff `(u, v)` is not a self-loop, and at least one of `(u, v)` and `(v, u)` exists in `G`.

For each edge `(u, v)` in `R`, `R[u][v]['capacity']` is equal to the capacity of `(u, v)` in `G` if it exists in `G` or zero otherwise. If the capacity is infinite, `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 `R.graph['inf']`. For each edge `(u, v)` in `R`, `R[u][v]['flow']` represents the flow function of `(u, v)` and satisfies `R[u][v]['flow'] == -R[v][u]['flow']`.

The flow value, defined as the total flow into `t`, the sink, is stored in `R.graph['flow_value']`. If `cutoff` is not specified, reachability to `t` using only edges `(u, v)` such that `R[u][v]['flow'] < R[u][v]['capacity']` induces a minimum `s-t` cut.

3.29.9 Network Simplex

network_simplex(G[, demand, capacity, weight])	Find a minimum cost flow satisfying all demands in digraph G.
min_cost_flow_cost(G[, demand, capacity, weight])	Find the cost of a minimum cost flow satisfying all demands in digraph G.
min_cost_flow(G[, demand, capacity, weight])	Returns a minimum cost flow satisfying all demands in digraph G.
cost_of_flow(G, flowDict[, weight])	Compute the cost of the flow given by flowDict on graph G.
max_flow_min_cost(G, s, t[, capacity, weight])	Returns a maximum (s, t)-flow of minimum cost.

network_simplex

network_simplex(*G*, *demand='demand'*, *capacity='capacity'*, *weight='weight'*)

Find a minimum cost flow satisfying all demands in digraph G.

This is a primal network simplex algorithm that uses the leaving arc rule to prevent cycling.

G is a digraph with edge costs and capacities and in which nodes have demand, i.e., they want to send or receive some amount of flow. A negative demand means that the node wants to send flow, a positive demand means that the node want to receive flow. A flow on the digraph G satisfies all demand if the net flow into each node is equal to the demand of that node.

Parameters

G
[NetworkX graph] DiGraph on which a minimum cost flow satisfying all demands is to be found.

demand
[string] Nodes of the graph G are expected to have an attribute demand that indicates how much flow a node wants to send (negative demand) or receive (positive demand). Note that the sum of the demands should be 0 otherwise the problem in not feasible. If this attribute is not present, a node is considered to have 0 demand. Default value: 'demand'.

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'.

weight
[string] Edges of the graph G are expected to have an attribute weight that indicates the cost incurred by sending one unit of flow on that edge. If not present, the weight is considered to be 0. Default value: 'weight'.

Returns

flowCost
[integer, float] Cost of a minimum cost flow satisfying all demands.

flowDict
[dictionary] Dictionary of dictionaries keyed by nodes such that flowDict[u][v] is the flow edge (u, v).

Raises

NetworkXError
This exception is raised if the input graph is not directed or not connected.

NetworkXUnfeasible
This exception is raised in the following situations:

- The sum of the demands is not zero. Then, there is no flow satisfying all demands.
- There is no flow satisfying all demand.

NetworkXUnbounded
This exception is raised if the digraph G has a cycle of negative cost and infinite capacity. Then, the cost of a flow satisfying all demands is unbounded below.

See also:

cost_of_flow, *max_flow_min_cost*, *min_cost_flow*, *min_cost_flow_cost*

Notes

This algorithm is not guaranteed to work if edge weights or demands are floating point numbers (overflows and roundoff errors can cause problems). As a workaround you can use integer numbers by multiplying the relevant edge attributes by a convenient constant factor (eg 100).

References

[1], [2]

Examples

A simple example of a min cost flow problem.

```
>>> G = nx.DiGraph()
>>> G.add_node("a", demand=-5)
>>> G.add_node("d", demand=5)
>>> G.add_edge("a", "b", weight=3, capacity=4)
>>> G.add_edge("a", "c", weight=6, capacity=10)
>>> G.add_edge("b", "d", weight=1, capacity=9)
>>> G.add_edge("c", "d", weight=2, capacity=5)
>>> flowCost, flowDict = nx.network_simplex(G)
>>> flowCost
24
>>> flowDict
{'a': {'b': 4, 'c': 1}, 'd': {}, 'b': {'d': 4}, 'c': {'d': 1}}
```

The mincost flow algorithm can also be used to solve shortest path problems. To find the shortest path between two nodes u and v, give all edges an infinite capacity, give node u a demand of -1 and node v a demand a 1. Then run the network simplex. The value of a min cost flow will be the distance between u and v and edges carrying positive flow will indicate the path.

```
>>> G = nx.DiGraph()
>>> G.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),
...     ]
... )
>>> G.add_node("s", demand=-1)
>>> G.add_node("v", demand=1)
>>> flowCost, flowDict = nx.network_simplex(G)
>>> flowCost == nx.shortest_path_length(G, "s", "v", weight="weight")
True
>>> sorted([(u, v) for u in flowDict for v in flowDict[u] if flowDict[u][v] > 0])
[('s', 'x'), ('u', 'v'), ('x', 'u')]
```

(continues on next page)

(continued from previous page)

```
>>> nx.shortest_path(G, "s", "v", weight="weight")
['s', 'x', 'u', 'v']
```

It is possible to change the name of the attributes used for the algorithm.

```
>>> G = nx.DiGraph()
>>> G.add_node("p", spam=-4)
>>> G.add_node("q", spam=2)
>>> G.add_node("a", spam=-2)
>>> G.add_node("d", spam=-1)
>>> G.add_node("t", spam=2)
>>> G.add_node("w", spam=3)
>>> G.add_edge("p", "q", cost=7, vacancies=5)
>>> G.add_edge("p", "a", cost=1, vacancies=4)
>>> G.add_edge("q", "d", cost=2, vacancies=3)
>>> G.add_edge("t", "q", cost=1, vacancies=2)
>>> G.add_edge("a", "t", cost=2, vacancies=4)
>>> G.add_edge("d", "w", cost=3, vacancies=4)
>>> G.add_edge("t", "w", cost=4, vacancies=1)
>>> flowCost, flowDict = nx.network_simplex(
...     G, demand="spam", capacity="vacancies", weight="cost"
... )
>>> flowCost
37
>>> flowDict
{'p': {'q': 2, 'a': 2}, 'q': {'d': 1}, 'a': {'t': 4}, 'd': {'w': 2}, 't': {'q': 1,
↪ 'w': 1}, 'w': {}}
```

min_cost_flow_cost

min_cost_flow_cost(*G*, *demand='demand'*, *capacity='capacity'*, *weight='weight'*)

Find the cost of a minimum cost flow satisfying all demands in digraph G.

G is a digraph with edge costs and capacities and in which nodes have demand, i.e., they want to send or receive some amount of flow. A negative demand means that the node wants to send flow, a positive demand means that the node want to receive flow. A flow on the digraph G satisfies all demand if the net flow into each node is equal to the demand of that node.

Parameters

G

[NetworkX graph] DiGraph on which a minimum cost flow satisfying all demands is to be found.

demand

[string] Nodes of the graph G are expected to have an attribute demand that indicates how much flow a node wants to send (negative demand) or receive (positive demand). Note that the sum of the demands should be 0 otherwise the problem in not feasible. If this attribute is not present, a node is considered to have 0 demand. Default value: 'demand'.

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'.

weight

[string] Edges of the graph G are expected to have an attribute weight that indicates the cost

incurred by sending one unit of flow on that edge. If not present, the weight is considered to be 0. Default value: 'weight'.

Returns

flowCost
[integer, float] Cost of a minimum cost flow satisfying all demands.

Raises

NetworkXError
This exception is raised if the input graph is not directed or not connected.

NetworkXUnfeasible
This exception is raised in the following situations:

- The sum of the demands is not zero. Then, there is no flow satisfying all demands.
- There is no flow satisfying all demand.

NetworkXUnbounded
This exception is raised if the digraph G has a cycle of negative cost and infinite capacity. Then, the cost of a flow satisfying all demands is unbounded below.

See also:

cost_of_flow, ***max_flow_min_cost***, ***min_cost_flow***, ***network_simplex***

Notes

This algorithm is not guaranteed to work if edge weights or demands are floating point numbers (overflows and roundoff errors can cause problems). As a workaround you can use integer numbers by multiplying the relevant edge attributes by a convenient constant factor (eg 100).

Examples

A simple example of a min cost flow problem.

```
>>> G = nx.DiGraph()
>>> G.add_node("a", demand=-5)
>>> G.add_node("d", demand=5)
>>> G.add_edge("a", "b", weight=3, capacity=4)
>>> G.add_edge("a", "c", weight=6, capacity=10)
>>> G.add_edge("b", "d", weight=1, capacity=9)
>>> G.add_edge("c", "d", weight=2, capacity=5)
>>> flowCost = nx.min_cost_flow_cost(G)
>>> flowCost
24
```

min_cost_flow

min_cost_flow(*G*, *demand='demand'*, *capacity='capacity'*, *weight='weight'*)

Returns a minimum cost flow satisfying all demands in digraph G.

G is a digraph with edge costs and capacities and in which nodes have demand, i.e., they want to send or receive some amount of flow. A negative demand means that the node wants to send flow, a positive demand means that the node want to receive flow. A flow on the digraph G satisfies all demand if the net flow into each node is equal to the demand of that node.

Parameters

G
[NetworkX graph] DiGraph on which a minimum cost flow satisfying all demands is to be found.

demand
[string] Nodes of the graph G are expected to have an attribute demand that indicates how much flow a node wants to send (negative demand) or receive (positive demand). Note that the sum of the demands should be 0 otherwise the problem in not feasible. If this attribute is not present, a node is considered to have 0 demand. Default value: 'demand'.

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'.

weight
[string] Edges of the graph G are expected to have an attribute weight that indicates the cost incurred by sending one unit of flow on that edge. If not present, the weight is considered to be 0. Default value: 'weight'.

Returns

flowDict
[dictionary] Dictionary of dictionaries keyed by nodes such that flowDict[u][v] is the flow edge (u, v).

Raises

NetworkXError
This exception is raised if the input graph is not directed or not connected.

NetworkXUnfeasible
This exception is raised in the following situations:

- The sum of the demands is not zero. Then, there is no flow satisfying all demands.
- There is no flow satisfying all demand.

NetworkXUnbounded
This exception is raised if the digraph G has a cycle of negative cost and infinite capacity. Then, the cost of a flow satisfying all demands is unbounded below.

See also:

cost_of_flow, *max_flow_min_cost*, *min_cost_flow_cost*, *network_simplex*

Notes

This algorithm is not guaranteed to work if edge weights or demands are floating point numbers (overflows and roundoff errors can cause problems). As a workaround you can use integer numbers by multiplying the relevant edge attributes by a convenient constant factor (eg 100).

Examples

A simple example of a min cost flow problem.

```
>>> G = nx.DiGraph()
>>> G.add_node("a", demand=-5)
>>> G.add_node("d", demand=5)
>>> G.add_edge("a", "b", weight=3, capacity=4)
>>> G.add_edge("a", "c", weight=6, capacity=10)
>>> G.add_edge("b", "d", weight=1, capacity=9)
>>> G.add_edge("c", "d", weight=2, capacity=5)
>>> flowDict = nx.min_cost_flow(G)
```

cost_of_flow

cost_of_flow(*G*, *flowDict*, *weight='weight'*)

Compute the cost of the flow given by flowDict on graph G.

Note that this function does not check for the validity of the flow flowDict. This function will fail if the graph G and the flow don't have the same edge set.

Parameters

G
: [NetworkX graph] DiGraph on which a minimum cost flow satisfying all demands is to be found.

weight
: [string] Edges of the graph G are expected to have an attribute weight that indicates the cost incurred by sending one unit of flow on that edge. If not present, the weight is considered to be 0. Default value: 'weight'.

flowDict
: [dictionary] Dictionary of dictionaries keyed by nodes such that flowDict[u][v] is the flow edge (u, v).

Returns

cost
: [Integer, float] The total cost of the flow. This is given by the sum over all edges of the product of the edge's flow and the edge's weight.

See also:

max_flow_min_cost, *min_cost_flow*, *min_cost_flow_cost*, *network_simplex*

Notes

This algorithm is not guaranteed to work if edge weights or demands are floating point numbers (overflows and roundoff errors can cause problems). As a workaround you can use integer numbers by multiplying the relevant edge attributes by a convenient constant factor (eg 100).

max_flow_min_cost

max_flow_min_cost(*G*, *s*, *t*, *capacity='capacity'*, *weight='weight'*)

Returns a maximum (s, t)-flow of minimum cost.

G is a digraph with edge costs and capacities. There is a source node s and a sink node t. This function finds a maximum flow from s to t whose total cost is minimized.

Parameters

G
[NetworkX graph] DiGraph on which a minimum cost flow satisfying all demands is to be found.

s: node label
Source of the flow.

t: node label
Destination of 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'.

weight: string
Edges of the graph G are expected to have an attribute weight that indicates the cost incurred by sending one unit of flow on that edge. If not present, the weight is considered to be 0. Default value: 'weight'.

Returns

flowDict: dictionary
Dictionary of dictionaries keyed by nodes such that flowDict[u][v] is the flow edge (u, v).

Raises

NetworkXError
This exception is raised if the input graph is not directed or not connected.

NetworkXUnbounded
This exception is raised if there is an infinite capacity path from s to t in G. In this case there is no maximum flow. This exception is also raised if the digraph G has a cycle of negative cost and infinite capacity. Then, the cost of a flow is unbounded below.

See also:

cost_of_flow, *min_cost_flow*, *min_cost_flow_cost*, *network_simplex*

Notes

This algorithm is not guaranteed to work if edge weights or demands are floating point numbers (overflows and roundoff errors can cause problems). As a workaround you can use integer numbers by multiplying the relevant edge attributes by a convenient constant factor (eg 100).

Examples

```
>>> G = nx.DiGraph()
>>> G.add_edges_from(
...     [
...         (1, 2, {"capacity": 12, "weight": 4}),
...         (1, 3, {"capacity": 20, "weight": 6}),
...         (2, 3, {"capacity": 6, "weight": -3}),
...         (2, 6, {"capacity": 14, "weight": 1}),
...         (3, 4, {"weight": 9}),
...         (3, 5, {"capacity": 10, "weight": 5}),
...         (4, 2, {"capacity": 19, "weight": 13}),
...         (4, 5, {"capacity": 4, "weight": 0}),
...         (5, 7, {"capacity": 28, "weight": 2}),
...         (6, 5, {"capacity": 11, "weight": 1}),
...         (6, 7, {"weight": 8}),
...         (7, 4, {"capacity": 6, "weight": 6}),
...     ]
... )
>>> mincostFlow = nx.max_flow_min_cost(G, 1, 7)
>>> mincost = nx.cost_of_flow(G, mincostFlow)
>>> mincost
373
>>> from networkx.algorithms.flow import maximum_flow
>>> maxFlow = maximum_flow(G, 1, 7)[1]
>>> nx.cost_of_flow(G, maxFlow) >= mincost
True
>>> mincostFlowValue = sum((mincostFlow[u][7] for u in G.predecessors(7))) - sum(
...     (mincostFlow[7][v] for v in G.successors(7))
... )
>>> mincostFlowValue == nx.maximum_flow_value(G, 1, 7)
True
```

3.29.10 Capacity Scaling Minimum Cost Flow

capacity_scaling(G[, demand, capacity, ...])	Find a minimum cost flow satisfying all demands in digraph G.

capacity_scaling

capacity_scaling(*G*, *demand='demand'*, *capacity='capacity'*, *weight='weight'*, *heap=<class 'networkx.utils.heaps.BinaryHeap'>*)

Find a minimum cost flow satisfying all demands in digraph G.

This is a capacity scaling successive shortest augmenting path algorithm.

G is a digraph with edge costs and capacities and in which nodes have demand, i.e., they want to send or receive some amount of flow. A negative demand means that the node wants to send flow, a positive demand means that the node want to receive flow. A flow on the digraph G satisfies all demand if the net flow into each node is equal to the demand of that node.

Parameters

G
[NetworkX graph] DiGraph or MultiDiGraph on which a minimum cost flow satisfying all demands is to be found.

demand
[string] Nodes of the graph G are expected to have an attribute demand that indicates how much flow a node wants to send (negative demand) or receive (positive demand). Note that the sum of the demands should be 0 otherwise the problem in not feasible. If this attribute is not present, a node is considered to have 0 demand. Default value: 'demand'.

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'.

weight
[string] Edges of the graph G are expected to have an attribute weight that indicates the cost incurred by sending one unit of flow on that edge. If not present, the weight is considered to be 0. Default value: 'weight'.

heap
[class] Type of heap to be used in the algorithm. It should be a subclass of `MinHeap` or implement a compatible interface.

If a stock heap implementation is to be used, `BinaryHeap` is recommended over `PairingHeap` for Python implementations without optimized attribute accesses (e.g., CPython) despite a slower asymptotic running time. For Python implementations with optimized attribute accesses (e.g., PyPy), `PairingHeap` provides better performance. Default value: `BinaryHeap`.

Returns

flowCost
[integer] Cost of a minimum cost flow satisfying all demands.

flowDict
[dictionary] If G is a digraph, a dict-of-dicts keyed by nodes such that flowDict[u][v] is the

flow on edge (u, v). If G is a MultiDiGraph, a dict-of-dicts-of-dicts keyed by nodes so that flowDict[u][v][key] is the flow on edge (u, v, key).

Raises

NetworkXError
This exception is raised if the input graph is not directed, not connected.

NetworkXUnfeasible
This exception is raised in the following situations:

- The sum of the demands is not zero. Then, there is no flow satisfying all demands.
- There is no flow satisfying all demand.

NetworkXUnbounded
This exception is raised if the digraph G has a cycle of negative cost and infinite capacity. Then, the cost of a flow satisfying all demands is unbounded below.

See also:

network_simplex()

Notes

This algorithm does not work if edge weights are floating-point numbers.

Examples

A simple example of a min cost flow problem.

```
>>> G = nx.DiGraph()
>>> G.add_node("a", demand=-5)
>>> G.add_node("d", demand=5)
>>> G.add_edge("a", "b", weight=3, capacity=4)
>>> G.add_edge("a", "c", weight=6, capacity=10)
>>> G.add_edge("b", "d", weight=1, capacity=9)
>>> G.add_edge("c", "d", weight=2, capacity=5)
>>> flowCost, flowDict = nx.capacity_scaling(G)
>>> flowCost
24
>>> flowDict
{'a': {'b': 4, 'c': 1}, 'd': {}, 'b': {'d': 4}, 'c': {'d': 1}}
```

It is possible to change the name of the attributes used for the algorithm.

```
>>> G = nx.DiGraph()
>>> G.add_node("p", spam=-4)
>>> G.add_node("q", spam=2)
>>> G.add_node("a", spam=-2)
>>> G.add_node("d", spam=-1)
>>> G.add_node("t", spam=2)
>>> G.add_node("w", spam=3)
>>> G.add_edge("p", "q", cost=7, vacancies=5)
>>> G.add_edge("p", "a", cost=1, vacancies=4)
>>> G.add_edge("q", "d", cost=2, vacancies=3)
>>> G.add_edge("t", "q", cost=1, vacancies=2)
```

(continues on next page)

(continued from previous page)

```
>>> G.add_edge("a", "t", cost=2, vacancies=4)
>>> G.add_edge("d", "w", cost=3, vacancies=4)
>>> G.add_edge("t", "w", cost=4, vacancies=1)
>>> flowCost, flowDict = nx.capacity_scaling(
...     G, demand="spam", capacity="vacancies", weight="cost"
... )
>>> flowCost
37
>>> flowDict
{'p': {'q': 2, 'a': 2}, 'q': {'d': 1}, 'a': {'t': 4}, 'd': {'w': 2}, 't': {'q': 1,
↪ 'w': 1}, 'w': {}}
```

3.30 Graph Hashing

Functions for hashing graphs to strings. Isomorphic graphs should be assigned identical hashes. For now, only Weisfeiler-Lehman hashing is implemented.

weisfeiler_lehman_graph_hash(G[, edge_attr, ...])	Return Weisfeiler Lehman (WL) graph hash.
weisfeiler_lehman_subgraph_hashes(G[, ...])	Return a dictionary of subgraph hashes by node.

3.30.1 weisfeiler_lehman_graph_hash

weisfeiler_lehman_graph_hash(*G*, *edge_attr=None*, *node_attr=None*, *iterations=3*, *digest_size=16*)

Return Weisfeiler Lehman (WL) graph hash.

The function iteratively aggregates and hashes neighbourhoods of each node. After each node's neighbors are hashed to obtain updated node labels, a hashed histogram of resulting labels is returned as the final hash.

Hashes are identical for isomorphic graphs and 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, default=None
The key in edge attribute dictionary to be used for hashing. If None, edge labels are ignored.

node_attr: string, 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.

iterations: int, default=3
Number of neighbor aggregations to perform. Should be larger for larger graphs.

digest_size: int, default=16
Size (in bits) of blake2b hash digest to use for hashing node labels.

Returns

h

[string] Hexadecimal string corresponding to hash of the input graph.

See also:

weisfeiler_lehman_subgraph_hashes

Notes

To return the WL hashes of each subgraph of a graph, use *weisfeiler_lehman_subgraph_hashes*

Similarity between hashes does not imply similarity between graphs.

References

[1]

Examples

Two graphs with edge attributes that are isomorphic, except for differences in the edge labels.

```
>>> G1 = nx.Graph()
>>> G1.add_edges_from(
...     [
...         (1, 2, {"label": "A"}),
...         (2, 3, {"label": "A"}),
...         (3, 1, {"label": "A"}),
...         (1, 4, {"label": "B"}),
...     ]
... )
>>> G2 = nx.Graph()
>>> G2.add_edges_from(
...     [
...         (5, 6, {"label": "B"}),
...         (6, 7, {"label": "A"}),
...         (7, 5, {"label": "A"}),
...         (7, 8, {"label": "A"}),
...     ]
... )
```

Omitting the `edge_attr` option, results in identical hashes.

```
>>> nx.weisfeiler_lehman_graph_hash(G1)
'7bc4dde9a09d0b94c5097b219891d81a'
>>> nx.weisfeiler_lehman_graph_hash(G2)
'7bc4dde9a09d0b94c5097b219891d81a'
```

With edge labels, the graphs are no longer assigned the same hash digest.

```
>>> nx.weisfeiler_lehman_graph_hash(G1, edge_attr="label")
'c653d85538bcf041d88c011f4f905f10'
>>> nx.weisfeiler_lehman_graph_hash(G2, edge_attr="label")
'3dcd84af1ca855d0eff3c978d88e7ec7'
```

3.30.2 weisfeiler_lehman_subgraph_hashes

weisfeiler_lehman_subgraph_hashes(*G*, *edge_attr=None*, *node_attr=None*, *iterations=3*, *digest_size=16*)

Return a dictionary of subgraph hashes by node.

The dictionary is keyed by node to a list of hashes in increasingly sized induced subgraphs containing the nodes within 2*k edges of the key node for increasing integer k until all nodes are included.

The function iteratively aggregates and hashes neighbourhoods of each node. This is achieved for each step by replacing for each node its label from the previous iteration with its hashed 1-hop neighborhood aggregate. The new node label is then appended to a list of node labels for each node.

To aggregate neighborhoods at each step for a node n, all labels of nodes adjacent to n are concatenated. If the `edge_attr` parameter is set, labels for each neighboring node are prefixed with the value of this attribute along the connecting edge from this neighbor to node n. The resulting string is then hashed to compress this information into a fixed digest size.

Thus, at the $i`thiterationnodeswithin : math : `2i$ distance influence any given hashed node label. We can therefore say that at depth i for node n we have a hash for a subgraph induced by the $2i$-hop neighborhood of n.

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, default=None
The key in edge attribute dictionary to be used for hashing. If None, edge labels are ignored.

node_attr: string, 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.

iterations: int, default=3
Number of neighbor aggregations to perform. Should be larger for larger graphs.

digest_size: int, default=16
Size (in bits) of blake2b hash digest to use for hashing node labels. The default size is 16 bits

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.

See also:

weisfeiler_lehman_graph_hash

Notes

To hash the full graph when subgraph hashes are not needed, use *weisfeiler_lehman_graph_hash* for efficiency.

Similarity between hashes does not imply similarity between graphs.

References

[1], [2]

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 4 hops around these nodes are isomorphic: each iteration aggregates 1-hop neighbourhoods meaning hashes at depth n are influenced by every node within $2n$ hops.

However the neighborhood of 6 hops is no longer isomorphic since their 3rd hash does not match.

These nodes may be candidates to be classified together since their local topology is similar.

3.31 Graphical degree sequence

Test sequences for graphiness.

is_graphical(sequence[, method])	Returns True if sequence is a valid degree sequence.
is_digraphical(in_sequence, out_sequence)	Returns True if some directed graph can realize the in- and out-degree sequences.
is_multigraphical(sequence)	Returns True if some multigraph can realize the sequence.
is_pseudographical(sequence)	Returns True if some pseudograph can realize the sequence.
is_valid_degree_sequence_havel_hakimi(...)	Returns True if deg_sequence can be realized by a simple graph.
is_valid_degree_sequence_erdos_gallai(...)	Returns True if deg_sequence can be realized by a simple graph.

3.31.1 is_graphical

is_graphical(*sequence*, *method='eg'*)

Returns True if sequence is a valid degree sequence.

A degree sequence is valid if some graph can realize it.

Parameters

sequence
[list or iterable container] A sequence of integer node degrees

method
["eg" | "hh" (default: 'eg')] The method used to validate the degree sequence. "eg" corresponds to the Erdős-Gallai algorithm [EG1960], [choudum1986], and "hh" to the Havel-Hakimi algorithm [havel1955], [hakimi1962], [CL1996].

Returns

valid
[bool] True if the sequence is a valid degree sequence and False if not.

References

[EG1960], [choudum1986], [havel1955], [hakimi1962], [CL1996]

Examples

```
>>> G = nx.path_graph(4)
>>> sequence = (d for n, d in G.degree())
>>> nx.is_graphical(sequence)
True
```

3.31.2 is_digraphical

is_digraphical(*in_sequence*, *out_sequence*)

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.

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]

3.31.3 is_multigraphical

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.

Notes

The worst-case run time is $O(n)$ where n is the length of the sequence.

References

[1]

3.31.4 is_pseudographical

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.

Notes

The worst-case run time is $O(n)$ where n is the length of the sequence.

References

[1]

3.31.5 is_valid_degree_sequence_havel_hakimi

is_valid_degree_sequence_havel_hakimi(*deg_sequence*)

Returns True if deg_sequence can be realized by a simple graph.

The validation proceeds using the Havel-Hakimi theorem [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.

Notes

The ZZ condition says that for the sequence d if

$$|d| >= \frac{(\max(d) + \min(d) + 1)^2}{4 * \min(d)}$$

then d is graphical. This was shown in Theorem 6 in [1].

References

[1], [havel1955], [hakimi1962], [CL1996]

3.31.6 is_valid_degree_sequence_erdos_gallai

is_valid_degree_sequence_erdos_gallai(*deg_sequence*)

Returns True if deg_sequence can be realized by a simple graph.

The validation is done using the Erdős-Gallai theorem [EG1960].

Parameters

deg_sequence
[list] A list of integers

Returns

valid
[bool] True if deg_sequence is graphical and False if not.

Notes

This implementation uses an equivalent form of the Erdős-Gallai criterion. Worst-case run time is $O(n)$ where n is the length of the sequence.

Specifically, a sequence d is graphical if and only if the sum of the sequence is even and for all strong indices k in the sequence,

$$\sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{j=k+1}^{n} \min(d_i, k) = k(n-1) - (k \sum_{j=0}^{k-1} n_j - \sum_{j=0}^{k-1} j n_j)$$

A strong index k is any index where d_k >= k and the value n_j is the number of occurrences of j in d. The maximal strong index is called the Durfee index.

This particular rearrangement comes from the proof of Theorem 3 in [2].

The ZZ condition says that for the sequence d if

$$|d| >= \frac{(\max(d) + \min(d) + 1)^2}{4 * \min(d)}$$

then d is graphical. This was shown in Theorem 6 in [2].

References

[1], [2], [EG1960]

3.32 Hierarchy

Flow Hierarchy.

flow_hierarchy(G[, weight])	Returns the flow hierarchy of a directed network.

3.32.1 flow_hierarchy

flow_hierarchy(*G*, *weight=None*)

Returns the flow hierarchy of a directed network.

Flow hierarchy is defined as the fraction of edges not participating in cycles in a directed graph [1].

Parameters

G
[DiGraph or MultiDiGraph] A directed graph

weight
[key,optional (default=None)] Attribute to use for node weights. If None the weight defaults to 1.

Returns

h
[float] Flow hierarchy value

Notes

The algorithm described in [1] computes the flow hierarchy through exponentiation of the adjacency matrix. This function implements an alternative approach that finds strongly connected components. An edge is in a cycle if and only if it is in a strongly connected component, which can be found in $O(m)$ time using Tarjan's algorithm.

References

[1]

3.33 Hybrid

Provides functions for finding and testing for locally `(k, l)`-connected graphs.

kl_connected_subgraph(G, k, l[, low_memory, ...])	Returns the maximum locally `(k, l)`-connected subgraph of `G`.
is_kl_connected(G, k, l[, low_memory])	Returns True if and only if `G` is locally `(k, l)`-connected.

3.33.1 kl_connected_subgraph

kl_connected_subgraph(*G*, *k*, *l*, *low_memory=False*, *same_as_graph=False*)

Returns the maximum locally `(k, l)`-connected subgraph of `G`.

A graph is locally `(k, l)`-connected if for each edge `(u, v)` in the graph there are at least `l` edge-disjoint paths of length at most `k` joining `u` to `v`.

Parameters

G
[NetworkX graph] The graph in which to find a maximum locally `(k, l)`-connected subgraph.

k
[integer] The maximum length of paths to consider. A higher number means a looser connectivity requirement.

l
[integer] The number of edge-disjoint paths. A higher number means a stricter connectivity requirement.

low_memory
[bool] If this is True, this function uses an algorithm that uses slightly more time but less memory.

same_as_graph
[bool] If True then return a tuple of the form `(H, is_same)`, where `H` is the maximum locally `(k, l)`-connected subgraph and `is_same` is a Boolean representing whether `G` is locally `(k, l)`-connected (and hence, whether `H` is simply a copy of the input graph `G`).

Returns

NetworkX graph or two-tuple
If `same_as_graph` is True, then this function returns a two-tuple as described above. Otherwise, it returns only the maximum locally `(k, l)`-connected subgraph.

See also:

is_kl_connected

References

[1]

3.33.2 is_kl_connected

is_kl_connected(*G*, *k*, *l*, *low_memory=False*)

Returns True if and only if `G` is locally `(k, l)`-connected.

A graph is locally `(k, l)`-connected if for each edge `(u, v)` in the graph there are at least `l` edge-disjoint paths of length at most `k` joining `u` to `v`.

Parameters

G
[NetworkX graph] The graph to test for local `(k, l)`-connectedness.

k
[integer] The maximum length of paths to consider. A higher number means a looser connectivity requirement.

l
[integer] The number of edge-disjoint paths. A higher number means a stricter connectivity requirement.

low_memory
[bool] If this is True, this function uses an algorithm that uses slightly more time but less memory.

Returns

bool
Whether the graph is locally `(k, l)`-connected subgraph.

See also:

kl_connected_subgraph

References

[1]

3.34 Isolates

Functions for identifying isolate (degree zero) nodes.

is_isolate(G, n)	Determines whether a node is an isolate.
isolates(G)	Iterator over isolates in the graph.
number_of_isolates(G)	Returns the number of isolates in the graph.

3.34.1 is_isolate

is_isolate(*G*, *n*)

Determines whether a node is an isolate.

An *isolate* is a node with no neighbors (that is, with degree zero). For directed graphs, this means no in-neighbors and no out-neighbors.

Parameters

G
[NetworkX graph]

n
[node] A node in `G`.

Returns

is_isolate
[bool] True if and only if `n` has no neighbors.

Examples

```
>>> G = nx.Graph()
>>> G.add_edge(1, 2)
>>> G.add_node(3)
>>> nx.is_isolate(G, 2)
False
>>> nx.is_isolate(G, 3)
True
```

3.34.2 isolates

isolates(*G*)

Iterator over isolates in the graph.

An *isolate* is a node with no neighbors (that is, with degree zero). For directed graphs, this means no in-neighbors and no out-neighbors.

Parameters

G
[NetworkX graph]

Returns

iterator
An iterator over the isolates of `G`.

Examples

To get a list of all isolates of a graph, use the `list` constructor:

```
>>> G = nx.Graph()
>>> G.add_edge(1, 2)
>>> G.add_node(3)
>>> list(nx.isolates(G))
[3]
```

To remove all isolates in the graph, first create a list of the isolates, then use `Graph.remove_nodes_from()`:

```
>>> G.remove_nodes_from(list(nx.isolates(G)))
>>> list(G)
[1, 2]
```

For digraphs, isolates have zero in-degree and zero out_degre:

```
>>> G = nx.DiGraph([(0, 1), (1, 2)])
>>> G.add_node(3)
>>> list(nx.isolates(G))
[3]
```

3.34.3 number_of_isolates

number_of_isolates(*G*)

Returns the number of isolates in the graph.

An *isolate* is a node with no neighbors (that is, with degree zero). For directed graphs, this means no in-neighbors and no out-neighbors.

Parameters

G
[NetworkX graph]

Returns

int
The number of degree zero nodes in the graph `G`.

3.35 Isomorphism

is_isomorphic(G1, G2[, node_match, edge_match])	Returns True if the graphs G1 and G2 are isomorphic and False otherwise.
could_be_isomorphic(G1, G2)	Returns False if graphs are definitely not isomorphic.
fast_could_be_isomorphic(G1, G2)	Returns False if graphs are definitely not isomorphic.
faster_could_be_isomorphic(G1, G2)	Returns False if graphs are definitely not isomorphic.

3.35.1 is_isomorphic

is_isomorphic(*G1*, *G2*, *node_match=None*, *edge_match=None*)

Returns True if the graphs G1 and G2 are isomorphic and False otherwise.

Parameters

G1, G2: graphs
The two graphs G1 and G2 must be 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 the isomorphism test. If node_match is not specified then node attributes are not considered.

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.

edge_match
[callable] A function that returns True if the edge attribute dictionary for the pair of nodes (u1, v1) in G1 and (u2, v2) in G2 should be considered equal during the isomorphism test. If edge_match is not specified then edge attributes are not considered.

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.

See also:

numerical_node_match, *numerical_edge_match*, *numerical_multiedge_match*
categorical_node_match, *categorical_edge_match*, *categorical_multiedge_match*

Notes

Uses the vf2 algorithm [1].

References

[1]

Examples

```
>>> import networkx.algorithms.isomorphism as iso
```

For digraphs G1 and G2, using 'weight' edge attribute (default: 1)

```
>>> G1 = nx.DiGraph()
>>> G2 = nx.DiGraph()
>>> nx.add_path(G1, [1, 2, 3, 4], weight=1)
>>> nx.add_path(G2, [10, 20, 30, 40], weight=2)
>>> em = iso.numerical_edge_match("weight", 1)
>>> nx.is_isomorphic(G1, G2)  # no weights considered
True
>>> nx.is_isomorphic(G1, G2, edge_match=em)  # match weights
False
```

For multidigraphs G1 and G2, using 'fill' node attribute (default: '')

```
>>> G1 = nx.MultiDiGraph()
>>> G2 = nx.MultiDiGraph()
>>> G1.add_nodes_from([1, 2, 3], fill="red")
>>> G2.add_nodes_from([10, 20, 30, 40], fill="red")
>>> nx.add_path(G1, [1, 2, 3, 4], weight=3, linewidth=2.5)
>>> nx.add_path(G2, [10, 20, 30, 40], weight=3)
>>> nm = iso.categorical_node_match("fill", "red")
>>> nx.is_isomorphic(G1, G2, node_match=nm)
True
```

For multidigraphs G1 and G2, using 'weight' edge attribute (default: 7)

```
>>> G1.add_edge(1, 2, weight=7)
1
>>> G2.add_edge(10, 20)
1
>>> em = iso.numerical_multiedge_match("weight", 7, rtol=1e-6)
>>> nx.is_isomorphic(G1, G2, edge_match=em)
True
```

For multigraphs G1 and G2, using 'weight' and 'linewidth' edge attributes with default values 7 and 2.5. Also using 'fill' node attribute with default value 'red'.

```
>>> em = iso.numerical_multiedge_match(["weight", "linewidth"], [7, 2.5])
>>> nm = iso.categorical_node_match("fill", "red")
>>> nx.is_isomorphic(G1, G2, edge_match=em, node_match=nm)
True
```

3.35.2 could_be_isomorphic

could_be_isomorphic(*G1*, *G2*)

Returns False if graphs are definitely not isomorphic. True does NOT guarantee isomorphism.

Parameters

G1, G2
[graphs] The two graphs G1 and G2 must be the same type.

Notes

Checks for matching degree, triangle, and number of cliques sequences. The triangle sequence contains the number of triangles each node is part of. The clique sequence contains for each node the number of maximal cliques involving that node.

3.35.3 fast_could_be_isomorphic

fast_could_be_isomorphic(*G1*, *G2*)

Returns False if graphs are definitely not isomorphic.

True does NOT guarantee isomorphism.

Parameters

G1, G2
[graphs] The two graphs G1 and G2 must be the same type.

Notes

Checks for matching degree and triangle sequences. The triangle sequence contains the number of triangles each node is part of.

3.35.4 faster_could_be_isomorphic

faster_could_be_isomorphic(*G1*, *G2*)

Returns False if graphs are definitely not isomorphic.

True does NOT guarantee isomorphism.

Parameters

G1, G2
[graphs] The two graphs G1 and G2 must be the same type.

Notes

Checks for matching degree sequences.

3.35.5 VF2++

VF2++ Algorithm

An implementation of the VF2++ algorithm for Graph Isomorphism testing.

The simplest interface to use this module is to call:

vf2pp_is_isomorphic: to check whether two graphs are isomorphic. *vf2pp_isomorphism*: to obtain the node mapping between two graphs, in case they are isomorphic. *vf2pp_all_isomorphisms*: to generate all possible mappings between two graphs, if isomorphic.

Introduction

The VF2++ algorithm, follows a similar logic to that of VF2, while also introducing new easy-to-check cutting rules and determining the optimal access order of nodes. It is also implemented in a non-recursive manner, which saves both time and space, when compared to its previous counterpart.

The optimal node ordering is obtained after taking into consideration both the degree but also the label rarity of each node. This way we place the nodes that are more likely to match, first in the order, thus examining the most promising branches in the beginning. The rules also consider node labels, making it easier to prune unfruitful branches early in the process.

Examples

Suppose G1 and G2 are Isomorphic Graphs. Verification is as follows:

Without node labels:

```
>>> import networkx as nx
>>> G1 = nx.path_graph(4)
>>> G2 = nx.path_graph(4)
>>> nx.vf2pp_is_isomorphic(G1, G2, node_label=None)
True
>>> nx.vf2pp_isomorphism(G1, G2, node_label=None)
{1: 1, 2: 2, 0: 0, 3: 3}
```

With node labels:

```
>>> G1 = nx.path_graph(4)
>>> G2 = nx.path_graph(4)
>>> mapped = {1: 1, 2: 2, 3: 3, 0: 0}
>>> nx.set_node_attributes(G1, dict(zip(G1, ["blue", "red", "green", "yellow"])),
↪"label")
>>> nx.set_node_attributes(G2, dict(zip([mapped[u] for u in G1], ["blue", "red",
↪"green", "yellow"])), "label")
>>> nx.vf2pp_is_isomorphic(G1, G2, node_label="label")
True
>>> nx.vf2pp_isomorphism(G1, G2, node_label="label")
{1: 1, 2: 2, 0: 0, 3: 3}
```

vf2pp_is_isomorphic(G1, G2[, node_label, ...])	Examines whether G1 and G2 are isomorphic.
vf2pp_all_isomorphisms(G1, G2[, node_label, ...])	Yields all the possible mappings between G1 and G2.
vf2pp_isomorphism(G1, G2[, node_label, ...])	Return an isomorphic mapping between `G1` and `G2` if it exists.

vf2pp_is_isomorphic

vf2pp_is_isomorphic(*G1*, *G2*, *node_label=None*, *default_label=None*)

Examines whether G1 and G2 are isomorphic.

Parameters

G1, G2
[NetworkX Graph or MultiGraph instances.] The two graphs to check for isomorphism.

node_label
[str, optional] The name of the node attribute to be used when comparing nodes. The default is `None`, meaning node attributes are not considered in the comparison. Any node that doesn't have the `node_label` attribute uses `default_label` instead.

default_label
[scalar] Default value to use when a node doesn't have an attribute named `node_label`. Default is `None`.

Returns

bool
True if the two graphs are isomorphic, False otherwise.

vf2pp_all_isomorphisms

vf2pp_all_isomorphisms(*G1*, *G2*, *node_label=None*, *default_label=None*)

Yields all the possible mappings between G1 and G2.

Parameters

G1, G2
[NetworkX Graph or MultiGraph instances.] The two graphs to check for isomorphism.

node_label
[str, optional] The name of the node attribute to be used when comparing nodes. The default is `None`, meaning node attributes are not considered in the comparison. Any node that doesn't have the `node_label` attribute uses `default_label` instead.

default_label
[scalar] Default value to use when a node doesn't have an attribute named `node_label`. Default is `None`.

Yields

dict
Isomorphic mapping between the nodes in `G1` and `G2`.

vf2pp_isomorphism

vf2pp_isomorphism(*G1*, *G2*, *node_label=None*, *default_label=None*)

Return an isomorphic mapping between `G1` and `G2` if it exists.

Parameters

G1, G2
[NetworkX Graph or MultiGraph instances.] The two graphs to check for isomorphism.

node_label
[str, optional] The name of the node attribute to be used when comparing nodes. The default is `None`, meaning node attributes are not considered in the comparison. Any node that doesn't have the `node_label` attribute uses `default_label` instead.

default_label
[scalar] Default value to use when a node doesn't have an attribute named `node_label`. Default is `None`.

Returns

dict or None
Node mapping if the two graphs are isomorphic. None otherwise.

3.35.6 Tree Isomorphism

An algorithm for finding if two undirected trees are isomorphic, and if so returns an isomorphism between the two sets of nodes.

This algorithm uses a routine to tell if two rooted trees (trees with a specified root node) are isomorphic, which may be independently useful.

This implements an algorithm from: The Design and Analysis of Computer Algorithms by Aho, Hopcroft, and Ullman Addison-Wesley Publishing 1974 Example 3.2 pp. 84-86.

A more understandable version of this algorithm is described in: Homework Assignment 5 McGill University SOCS 308-250B, Winter 2002 by Matthew Suderman http://crypto.cs.mcgill.ca/~crepeau/CS250/2004/HW5+.pdf

rooted_tree_isomorphism(t1, root1, t2, root2)	Given two rooted trees `t1` and `t2`, with roots `root1` and `root2` respectivly this routine will determine if they are isomorphic.
tree_isomorphism(t1, t2)	Given two undirected (or free) trees `t1` and `t2`, this routine will determine if they are isomorphic.

rooted_tree_isomorphism

rooted_tree_isomorphism(*t1*, *root1*, *t2*, *root2*)

Given two rooted trees `t1` and `t2`, with roots `root1` and `root2` respectivly this routine will determine if they are isomorphic.

These trees may be either directed or undirected, but if they are directed, all edges should flow from the root.

It returns the isomorphism, a mapping of the nodes of `t1` onto the nodes of `t2`, such that two trees are then identical.

Note that two trees may have more than one isomorphism, and this routine just returns one valid mapping.

Parameters

`t1`
[NetworkX graph] One of the trees being compared

`root1`
[a node of `t1` which is the root of the tree]

`t2`
[undirected NetworkX graph] The other tree being compared

`root2`
[a node of `t2` which is the root of the tree]

This is a subroutine used to implement `tree_isomorphism`, but will be somewhat faster if you already have rooted trees.

Returns

isomorphism
[list] A list of pairs in which the left element is a node in `t1` and the right element is a node in `t2`. The pairs are in arbitrary order. If the nodes in one tree is mapped to the names in the other, then trees will be identical. Note that an isomorphism will not necessarily be unique.

If `t1` and `t2` are not isomorphic, then it returns the empty list.

tree_isomorphism

`tree_isomorphism`(*t1*, *t2*)

Given two undirected (or free) trees `t1` and `t2`, this routine will determine if they are isomorphic. It returns the isomorphism, a mapping of the nodes of `t1` onto the nodes of `t2`, such that two trees are then identical.

Note that two trees may have more than one isomorphism, and this routine just returns one valid mapping.

Parameters

t1
[undirected NetworkX graph] One of the trees being compared

t2
[undirected NetworkX graph] The other tree being compared

Returns

isomorphism
[list] A list of pairs in which the left element is a node in `t1` and the right element is a node in `t2`. The pairs are in arbitrary order. If the nodes in one tree is mapped to the names in the other, then trees will be identical. Note that an isomorphism will not necessarily be unique.

If `t1` and `t2` are not isomorphic, then it returns the empty list.

Notes

This runs in O(n*log(n)) time for trees with n nodes.

3.35.7 Advanced Interfaces

VF2 Algorithm

An implementation of VF2 algorithm for graph isomorphism testing.

The simplest interface to use this module is to call networkx.is_isomorphic().

Introduction

The GraphMatcher and DiGraphMatcher are responsible for matching graphs or directed graphs in a predetermined manner. This usually means a check for an isomorphism, though other checks are also possible. For example, a subgraph of one graph can be checked for isomorphism to a second graph.

Matching is done via syntactic feasibility. It is also possible to check for semantic feasibility. Feasibility, then, is defined as the logical AND of the two functions.

To include a semantic check, the (Di)GraphMatcher class should be subclassed, and the semantic_feasibility() function should be redefined. By default, the semantic feasibility function always returns True. The effect of this is that semantics are not considered in the matching of G1 and G2.

Examples

Suppose G1 and G2 are isomorphic graphs. Verification is as follows:

```
>>> from networkx.algorithms import isomorphism
>>> G1 = nx.path_graph(4)
>>> G2 = nx.path_graph(4)
>>> GM = isomorphism.GraphMatcher(G1, G2)
>>> GM.is_isomorphic()
True
```

GM.mapping stores the isomorphism mapping from G1 to G2.

```
>>> GM.mapping
{0: 0, 1: 1, 2: 2, 3: 3}
```

Suppose G1 and G2 are isomorphic directed graphs. Verification is as follows:

```
>>> G1 = nx.path_graph(4, create_using=nx.DiGraph())
>>> G2 = nx.path_graph(4, create_using=nx.DiGraph())
>>> DiGM = isomorphism.DiGraphMatcher(G1, G2)
>>> DiGM.is_isomorphic()
True
```

DiGM.mapping stores the isomorphism mapping from G1 to G2.

```
>>> DiGM.mapping
{0: 0, 1: 1, 2: 2, 3: 3}
```

Subgraph Isomorphism

Graph theory literature can be ambiguous about the meaning of the above statement, and we seek to clarify it now.

In the VF2 literature, a mapping M is said to be a graph-subgraph isomorphism iff M is an isomorphism between G2 and a subgraph of G1. Thus, to say that G1 and G2 are graph-subgraph isomorphic is to say that a subgraph of G1 is isomorphic to G2.

Other literature uses the phrase 'subgraph isomorphic' as in 'G1 does not have a subgraph isomorphic to G2'. Another use is as an in adverb for isomorphic. Thus, to say that G1 and G2 are subgraph isomorphic is to say that a subgraph of G1 is isomorphic to G2.

Finally, the term 'subgraph' can have multiple meanings. In this context, 'subgraph' always means a 'node-induced subgraph'. Edge-induced subgraph isomorphisms are not directly supported, but one should be able to perform the check by making use of nx.line_graph(). For subgraphs which are not induced, the term 'monomorphism' is preferred over 'isomorphism'.

Let G=(N,E) be a graph with a set of nodes N and set of edges E.

If G'=(N',E') is a subgraph, then:
N' is a subset of N E' is a subset of E

If G'=(N',E') is a node-induced subgraph, then:
N' is a subset of N E' is the subset of edges in E relating nodes in N'

If G'=(N',E') is an edge-induced subgraph, then:
N' is the subset of nodes in N related by edges in E' E' is a subset of E

If G'=(N',E') is a monomorphism, then:
N' is a subset of N E' is a subset of the set of edges in E relating nodes in N'

Note that if G' is a node-induced subgraph of G, then it is always a subgraph monomorphism of G, but the opposite is not always true, as a monomorphism can have fewer edges.

References

[1] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, Mario Vento,
"A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs", IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 26, no. 10, pp. 1367-1372, Oct., 2004. http://ieeexplore.ieee.org/iel5/34/29305/01323804.pdf

[2] L. P. Cordella, P. Foggia, C. Sansone, M. Vento, "An Improved
Algorithm for Matching Large Graphs", 3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition, Cuen, pp. 149-159, 2001. https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.101.5342

See Also

syntactic_feasibility(), semantic_feasibility()

Notes

The implementation handles both directed and undirected graphs as well as multigraphs.

In general, the subgraph isomorphism problem is NP-complete whereas the graph isomorphism problem is most likely not NP-complete (although no polynomial-time algorithm is known to exist).

Graph Matcher

GraphMatcher.__init__(G1, G2[, node_match, ...])	Initialize graph matcher.
GraphMatcher.initialize()	Reinitializes the state of the algorithm.
GraphMatcher.is_isomorphic()	Returns True if G1 and G2 are isomorphic graphs.
GraphMatcher.subgraph_is_isomorphic()	Returns True if a subgraph of G1 is isomorphic to G2.
GraphMatcher.isomorphisms_iter()	Generator over isomorphisms between G1 and G2.
GraphMatcher.subgraph_isomorphisms_iter()	Generator over isomorphisms between a subgraph of G1 and G2.
GraphMatcher.candidate_pairs_iter()	Iterator over candidate pairs of nodes in G1 and G2.
GraphMatcher.match()	Extends the isomorphism mapping.
GraphMatcher.semantic_feasibility(G1_node, ...)	Returns True if mapping G1_node to G2_node is semantically feasible.
GraphMatcher.syntactic_feasibility(G1_node, ...)	Returns True if adding (G1_node, G2_node) is syntactically feasible.

GraphMatcher.__init__

GraphMatcher.**__init__**(*G1*, *G2*, *node_match=None*, *edge_match=None*)

Initialize graph matcher.

Parameters

G1, G2: graph
The graphs to be tested.

node_match: callable
A function that returns True iff node n1 in G1 and n2 in G2 should be considered equal during the isomorphism test. The function will be called like:

```
node_match(G1.nodes[n1], G2.nodes[n2])
```

That is, the function will receive the node attribute dictionaries of the nodes under consideration. If None, then no attributes are considered when testing for an isomorphism.

edge_match: callable
A function that returns True iff the edge attribute dictionary for the pair of nodes (u1, v1) in G1 and (u2, v2) in G2 should be considered equal during the isomorphism test. 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. If None, then no attributes are considered when testing for an isomorphism.

GraphMatcher.initialize

GraphMatcher.**initialize**()

Reinitializes the state of the algorithm.

This method should be redefined if using something other than GMState. If only subclassing GraphMatcher, a redefinition is not necessary.

GraphMatcher.is_isomorphic

GraphMatcher.**is_isomorphic**()

Returns True if G1 and G2 are isomorphic graphs.

GraphMatcher.subgraph_is_isomorphic

GraphMatcher.**subgraph_is_isomorphic**()

Returns True if a subgraph of G1 is isomorphic to G2.

GraphMatcher.isomorphisms_iter

GraphMatcher.**isomorphisms_iter**()

Generator over isomorphisms between G1 and G2.

GraphMatcher.subgraph_isomorphisms_iter

GraphMatcher.**subgraph_isomorphisms_iter**()

Generator over isomorphisms between a subgraph of G1 and G2.

GraphMatcher.candidate_pairs_iter

GraphMatcher.**candidate_pairs_iter**()

Iterator over candidate pairs of nodes in G1 and G2.

GraphMatcher.match

GraphMatcher.**match**()

Extends the isomorphism mapping.

This function is called recursively to determine if a complete isomorphism can be found between G1 and G2. It cleans up the class variables after each recursive call. If an isomorphism is found, we yield the mapping.

GraphMatcher.semantic_feasibility

GraphMatcher.**semantic_feasibility**(*G1_node*, *G2_node*)

Returns True if mapping G1_node to G2_node is semantically feasible.

GraphMatcher.syntactic_feasibility

GraphMatcher.**syntactic_feasibility**(*G1_node*, *G2_node*)

Returns True if adding (G1_node, G2_node) is syntactically feasible.

This function returns True if it is adding the candidate pair to the current partial isomorphism/monomorphism mapping is allowable. The addition is allowable if the inclusion of the candidate pair does not make it impossible for an isomorphism/monomorphism to be found.

DiGraph Matcher

DiGraphMatcher.__init__(G1, G2[, ...])	Initialize graph matcher.
DiGraphMatcher.initialize()	Reinitializes the state of the algorithm.
DiGraphMatcher.is_isomorphic()	Returns True if G1 and G2 are isomorphic graphs.
DiGraphMatcher.subgraph_is_isomorphic()	Returns True if a subgraph of G1 is isomorphic to G2.
DiGraphMatcher.isomorphisms_iter()	Generator over isomorphisms between G1 and G2.
DiGraphMatcher.subgraph_isomorphisms_iter()	Generator over isomorphisms between a subgraph of G1 and G2.
DiGraphMatcher.candidate_pairs_iter()	Iterator over candidate pairs of nodes in G1 and G2.
DiGraphMatcher.match()	Extends the isomorphism mapping.
DiGraphMatcher.semantic_feasibility(G1_node, ...)	Returns True if mapping G1_node to G2_node is semantically feasible.
DiGraphMatcher.syntactic_feasibility(...)	Returns True if adding (G1_node, G2_node) is syntactically feasible.

DiGraphMatcher.__init__

DiGraphMatcher.**__init__**(*G1*, *G2*, *node_match=None*, *edge_match=None*)

Initialize graph matcher.

Parameters

G1, G2
[graph] The graphs to be tested.

node_match
[callable] A function that returns True iff node n1 in G1 and n2 in G2 should be considered equal during the isomorphism test. The function will be called like:

```
node_match(G1.nodes[n1], G2.nodes[n2])
```

That is, the function will receive the node attribute dictionaries of the nodes under consideration. If None, then no attributes are considered when testing for an isomorphism.

edge_match
[callable] A function that returns True iff the edge attribute dictionary for the pair of nodes

(u1, v1) in G1 and (u2, v2) in G2 should be considered equal during the isomorphism test. 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. If None, then no attributes are considered when testing for an isomorphism.

DiGraphMatcher.initialize

DiGraphMatcher.**initialize**()

Reinitializes the state of the algorithm.

This method should be redefined if using something other than DiGMState. If only subclassing GraphMatcher, a redefinition is not necessary.

DiGraphMatcher.is_isomorphic

DiGraphMatcher.**is_isomorphic**()

Returns True if G1 and G2 are isomorphic graphs.

DiGraphMatcher.subgraph_is_isomorphic

DiGraphMatcher.**subgraph_is_isomorphic**()

Returns True if a subgraph of G1 is isomorphic to G2.

DiGraphMatcher.isomorphisms_iter

DiGraphMatcher.**isomorphisms_iter**()

Generator over isomorphisms between G1 and G2.

DiGraphMatcher.subgraph_isomorphisms_iter

DiGraphMatcher.**subgraph_isomorphisms_iter**()

Generator over isomorphisms between a subgraph of G1 and G2.

DiGraphMatcher.candidate_pairs_iter

DiGraphMatcher.**candidate_pairs_iter**()

Iterator over candidate pairs of nodes in G1 and G2.

DiGraphMatcher.match

DiGraphMatcher.**match**()

Extends the isomorphism mapping.

This function is called recursively to determine if a complete isomorphism can be found between G1 and G2. It cleans up the class variables after each recursive call. If an isomorphism is found, we yield the mapping.

DiGraphMatcher.semantic_feasibility

DiGraphMatcher.**semantic_feasibility**(*G1_node*, *G2_node*)

Returns True if mapping G1_node to G2_node is semantically feasible.

DiGraphMatcher.syntactic_feasibility

DiGraphMatcher.**syntactic_feasibility**(*G1_node*, *G2_node*)

Returns True if adding (G1_node, G2_node) is syntactically feasible.

This function returns True if it is adding the candidate pair to the current partial isomorphism/monomorphism mapping is allowable. The addition is allowable if the inclusion of the candidate pair does not make it impossible for an isomorphism/monomorphism to be found.

Match helpers

categorical_node_match(attr, default)	Returns a comparison function for a categorical node attribute.
categorical_edge_match(attr, default)	Returns a comparison function for a categorical edge attribute.
categorical_multiedge_match(attr, default)	Returns a comparison function for a categorical edge attribute.
numerical_node_match(attr, default[, rtol, atol])	Returns a comparison function for a numerical node attribute.
numerical_edge_match(attr, default[, rtol, atol])	Returns a comparison function for a numerical edge attribute.
numerical_multiedge_match(attr, default[, ...])	Returns a comparison function for a numerical edge attribute.
generic_node_match(attr, default, op)	Returns a comparison function for a generic attribute.
generic_edge_match(attr, default, op)	Returns a comparison function for a generic attribute.
generic_multiedge_match(attr, default, op)	Returns a comparison function for a generic attribute.

categorical_node_match

categorical_node_match (*attr*, *default*)

Returns a comparison function for a categorical node attribute.

The value(s) of the attr(s) must be hashable and comparable via the == operator since they are placed into a set([]) object. If the sets from G1 and G2 are the same, then the constructed function returns True.

Parameters

attr
[string | list] The categorical node attribute to compare, or a list of categorical node attributes to compare.

default
[value | list] The default value for the categorical node attribute, or a list of default values for the categorical node attributes.

Returns

match
[function] The customized, categorical `node_match` function.

Examples

```
>>> import networkx.algorithms.isomorphism as iso
>>> nm = iso.categorical_node_match("size", 1)
>>> nm = iso.categorical_node_match(["color", "size"], ["red", 2])
```

categorical_edge_match

categorical_edge_match (*attr*, *default*)

Returns a comparison function for a categorical edge attribute.

The value(s) of the attr(s) must be hashable and comparable via the == operator since they are placed into a set([]) object. If the sets from G1 and G2 are the same, then the constructed function returns True.

Parameters

attr
[string | list] The categorical edge attribute to compare, or a list of categorical edge attributes to compare.

default
[value | list] The default value for the categorical edge attribute, or a list of default values for the categorical edge attributes.

Returns

match
[function] The customized, categorical `edge_match` function.

Examples

```
>>> import networkx.algorithms.isomorphism as iso
>>> nm = iso.categorical_edge_match("size", 1)
>>> nm = iso.categorical_edge_match(["color", "size"], ["red", 2])
```

categorical_multiedge_match

categorical_multiedge_match(*attr*, *default*)

Returns a comparison function for a categorical edge attribute.

The value(s) of the attr(s) must be hashable and comparable via the == operator since they are placed into a set([]) object. If the sets from G1 and G2 are the same, then the constructed function returns True.

Parameters

attr
[string | list] The categorical edge attribute to compare, or a list of categorical edge attributes to compare.

default
[value | list] The default value for the categorical edge attribute, or a list of default values for the categorical edge attributes.

Returns

match
[function] The customized, categorical `edge_match` function.

Examples

```
>>> import networkx.algorithms.isomorphism as iso
>>> nm = iso.categorical_multiedge_match("size", 1)
>>> nm = iso.categorical_multiedge_match(["color", "size"], ["red", 2])
```

numerical_node_match

numerical_node_match(*attr*, *default*, *rtol=1e-05*, *atol=1e-08*)

Returns a comparison function for a numerical node attribute.

The value(s) of the attr(s) must be numerical and sortable. If the sorted list of values from G1 and G2 are the same within some tolerance, then the constructed function returns True.

Parameters

attr
[string | list] The numerical node attribute to compare, or a list of numerical node attributes to compare.

default
[value | list] The default value for the numerical node attribute, or a list of default values for the numerical node attributes.

rtol
[float] The relative error tolerance.

atol
[float] The absolute error tolerance.

Returns

match
[function] The customized, numerical `node_match` function.

Examples

```
>>> import networkx.algorithms.isomorphism as iso
>>> nm = iso.numerical_node_match("weight", 1.0)
>>> nm = iso.numerical_node_match(["weight", "linewidth"], [0.25, 0.5])
```

numerical_edge_match

numerical_edge_match(*attr*, *default*, *rtol=1e-05*, *atol=1e-08*)

Returns a comparison function for a numerical edge attribute.

The value(s) of the attr(s) must be numerical and sortable. If the sorted list of values from G1 and G2 are the same within some tolerance, then the constructed function returns True.

Parameters

attr
[string | list] The numerical edge attribute to compare, or a list of numerical edge attributes to compare.

default
[value | list] The default value for the numerical edge attribute, or a list of default values for the numerical edge attributes.

rtol
[float] The relative error tolerance.

atol
[float] The absolute error tolerance.

Returns

match
[function] The customized, numerical `edge_match` function.

Examples

```
>>> import networkx.algorithms.isomorphism as iso
>>> nm = iso.numerical_edge_match("weight", 1.0)
>>> nm = iso.numerical_edge_match(["weight", "linewidth"], [0.25, 0.5])
```

numerical_multiedge_match

numerical_multiedge_match(*attr*, *default*, *rtol=1e-05*, *atol=1e-08*)

Returns a comparison function for a numerical edge attribute.

The value(s) of the attr(s) must be numerical and sortable. If the sorted list of values from G1 and G2 are the same within some tolerance, then the constructed function returns True.

Parameters

attr
[string | list] The numerical edge attribute to compare, or a list of numerical edge attributes to compare.

default
[value | list] The default value for the numerical edge attribute, or a list of default values for the numerical edge attributes.

rtol
[float] The relative error tolerance.

atol
[float] The absolute error tolerance.

Returns

match
[function] The customized, numerical `edge_match` function.

Examples

```
>>> import networkx.algorithms.isomorphism as iso
>>> nm = iso.numerical_multiedge_match("weight", 1.0)
>>> nm = iso.numerical_multiedge_match(["weight", "linewidth"], [0.25, 0.5])
```

generic_node_match

generic_node_match(*attr*, *default*, *op*)

Returns a comparison function for a generic attribute.

The value(s) of the attr(s) are compared using the specified operators. If all the attributes are equal, then the constructed function returns True.

Parameters

attr
[string | list] The node attribute to compare, or a list of node attributes to compare.

default
[value | list] The default value for the node attribute, or a list of default values for the node attributes.

op
[callable | list] The operator to use when comparing attribute values, or a list of operators to use when comparing values for each attribute.

Returns

match
[function] The customized, generic `node_match` function.

Examples

```
>>> from operator import eq
>>> from math import isclose
>>> from networkx.algorithms.isomorphism import generic_node_match
>>> nm = generic_node_match("weight", 1.0, isclose)
>>> nm = generic_node_match("color", "red", eq)
>>> nm = generic_node_match(["weight", "color"], [1.0, "red"], [isclose, eq])
```

generic_edge_match

generic_edge_match(*attr*, *default*, *op*)

Returns a comparison function for a generic attribute.

The value(s) of the attr(s) are compared using the specified operators. If all the attributes are equal, then the constructed function returns True.

Parameters

attr
[string | list] The edge attribute to compare, or a list of edge attributes to compare.

default
[value | list] The default value for the edge attribute, or a list of default values for the edge attributes.

op
[callable | list] The operator to use when comparing attribute values, or a list of operators to use when comparing values for each attribute.

Returns

match
[function] The customized, generic `edge_match` function.

Examples

```
>>> from operator import eq
>>> from math import isclose
>>> from networkx.algorithms.isomorphism import generic_edge_match
>>> nm = generic_edge_match("weight", 1.0, isclose)
>>> nm = generic_edge_match("color", "red", eq)
>>> nm = generic_edge_match(["weight", "color"], [1.0, "red"], [isclose, eq])
```

generic_multiedge_match

generic_multiedge_match(*attr*, *default*, *op*)

Returns a comparison function for a generic attribute.

The value(s) of the attr(s) are compared using the specified operators. If all the attributes are equal, then the constructed function returns True. Potentially, the constructed edge_match function can be slow since it must verify that no isomorphism exists between the multiedges before it returns False.

Parameters

attr
[string | list] The edge attribute to compare, or a list of node attributes to compare.

default
[value | list] The default value for the edge attribute, or a list of default values for the dgeattributes.

op
[callable | list] The operator to use when comparing attribute values, or a list of operators to use when comparing values for each attribute.

Returns

match
[function] The customized, generic `edge_match` function.

Examples

```
>>> from operator import eq
>>> from math import isclose
>>> from networkx.algorithms.isomorphism import generic_node_match
>>> nm = generic_node_match("weight", 1.0, isclose)
>>> nm = generic_node_match("color", "red", eq)
>>> nm = generic_node_match(["weight", "color"], [1.0, "red"], [isclose, eq])
...
```

ISMAGS Algorithm

Provides a Python implementation of the ISMAGS algorithm. [1]

It is capable of finding (subgraph) isomorphisms between two graphs, taking the symmetry of the subgraph into account. In most cases the VF2 algorithm is faster (at least on small graphs) than this implementation, but in some cases there is an exponential number of isomorphisms that are symmetrically equivalent. In that case, the ISMAGS algorithm will provide only one solution per symmetry group.

```
>>> petersen = nx.petersen_graph()
>>> ismags = nx.isomorphism.ISMAGS(petersen, petersen)
>>> isomorphisms = list(ismags.isomorphisms_iter(symmetry=False))
>>> len(isomorphisms)
120
>>> isomorphisms = list(ismags.isomorphisms_iter(symmetry=True))
>>> answer = [{0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6, 7: 7, 8: 8, 9: 9}]
>>> answer == isomorphisms
True
```

In addition, this implementation also provides an interface to find the largest common induced subgraph [2] between any two graphs, again taking symmetry into account. Given `graph` and `subgraph` the algorithm will remove nodes from the `subgraph` until `subgraph` is isomorphic to a subgraph of `graph`. Since only the symmetry of `subgraph` is taken into account it is worth thinking about how you provide your graphs:

```
>>> graph1 = nx.path_graph(4)
>>> graph2 = nx.star_graph(3)
>>> ismags = nx.isomorphism.ISMAGS(graph1, graph2)
>>> ismags.is_isomorphic()
False
>>> largest_common_subgraph = list(ismags.largest_common_subgraph())
>>> answer = [{1: 0, 0: 1, 2: 2}, {2: 0, 1: 1, 3: 2}]
>>> answer == largest_common_subgraph
True
>>> ismags2 = nx.isomorphism.ISMAGS(graph2, graph1)
>>> largest_common_subgraph = list(ismags2.largest_common_subgraph())
>>> answer = [
...     {1: 0, 0: 1, 2: 2},
...     {1: 0, 0: 1, 3: 2},
...     {2: 0, 0: 1, 1: 2},
...     {2: 0, 0: 1, 3: 2},
...     {3: 0, 0: 1, 1: 2},
...     {3: 0, 0: 1, 2: 2},
... ]
>>> answer == largest_common_subgraph
True
```

However, when not taking symmetry into account, it doesn't matter:

```
>>> largest_common_subgraph = list(ismags.largest_common_subgraph(symmetry=False))
>>> answer = [
...     {1: 0, 0: 1, 2: 2},
...     {1: 0, 2: 1, 0: 2},
...     {2: 0, 1: 1, 3: 2},
...     {2: 0, 3: 1, 1: 2},
...     {1: 0, 0: 1, 2: 3},
...     {1: 0, 2: 1, 0: 3},
...     {2: 0, 1: 1, 3: 3},
...     {2: 0, 3: 1, 1: 3},
...     {1: 0, 0: 2, 2: 3},
...     {1: 0, 2: 2, 0: 3},
...     {2: 0, 1: 2, 3: 3},
...     {2: 0, 3: 2, 1: 3},
... ]
>>> answer == largest_common_subgraph
True
>>> largest_common_subgraph = list(ismags2.largest_common_subgraph(symmetry=False))
>>> answer = [
...     {1: 0, 0: 1, 2: 2},
...     {1: 0, 0: 1, 3: 2},
...     {2: 0, 0: 1, 1: 2},
...     {2: 0, 0: 1, 3: 2},
...     {3: 0, 0: 1, 1: 2},
...     {3: 0, 0: 1, 2: 2},
...     {1: 1, 0: 2, 2: 3},
...     {1: 1, 0: 2, 3: 3},
...     {2: 1, 0: 2, 1: 3},
...     {2: 1, 0: 2, 3: 3},
```

(continues on next page)

(continued from previous page)

```
...      {3: 1, 0: 2, 1: 3},
...      {3: 1, 0: 2, 2: 3},
... ]
>>> answer == largest_common_subgraph
True
```

Notes

- The current implementation works for undirected graphs only. The algorithm in general should work for directed graphs as well though.
- Node keys for both provided graphs need to be fully orderable as well as hashable.
- Node and edge equality is assumed to be transitive: if A is equal to B, and B is equal to C, then A is equal to C.

References

ISMAGS object

ISMAGS(graph, subgraph[, node_match, ...])	Implements the ISMAGS subgraph matching algorith.

networkx.algorithms.isomorphism.ISMAGS

class ISMAGS(*graph*, *subgraph*, *node_match=None*, *edge_match=None*, *cache=None*)

Implements the ISMAGS subgraph matching algorith. [1] ISMAGS stands for "Index-based Subgraph Matching Algorithm with General Symmetries". As the name implies, it is symmetry aware and will only generate non-symmetric isomorphisms.

Notes

The implementation imposes additional conditions compared to the VF2 algorithm on the graphs provided and the comparison functions (`node_equality` and `edge_equality`):

- Node keys in both graphs must be orderable as well as hashable.
- Equality must be transitive: if A is equal to B, and B is equal to C, then A must be equal to C.

References

[1]

Attributes

graph: networkx.Graph
subgraph: networkx.Graph
node_equality: collections.abc.Callable
The function called to see if two nodes should be considered equal. It's signature looks like this: `f(graph1: networkx.Graph, node1, graph2: networkx.Graph, node2) -> bool`. `node1` is a node in `graph1`, and `node2` a node in `graph2`. Constructed from the argument `node_match`.

edge_equality: collections.abc.Callable
The function called to see if two edges should be considered equal. It's signature looks like this: `f(graph1: networkx.Graph, edge1, graph2: networkx.Graph, edge2) -> bool`. `edge1` is an edge in `graph1`, and `edge2` an edge in `graph2`. Constructed from the argument `edge_match`.

__init__(*graph*, *subgraph*, *node_match=None*, *edge_match=None*, *cache=None*)

Parameters

graph: networkx.Graph
subgraph: networkx.Graph
node_match: collections.abc.Callable or None
Function used to determine whether two nodes are equivalent. Its signature should look like `f(n1: dict, n2: dict) -> bool`, with `n1` and `n2` node property dicts. See also *categorical_node_match()* and friends. If `None`, all nodes are considered equal.

edge_match: collections.abc.Callable or None
Function used to determine whether two edges are equivalent. Its signature should look like `f(e1: dict, e2: dict) -> bool`, with `e1` and `e2` edge property dicts. See also *categorical_edge_match()* and friends. If `None`, all edges are considered equal.

cache: collections.abc.Mapping
A cache used for caching graph symmetries.

Methods

analyze_symmetry(graph, node_partitions, ...)	Find a minimal set of permutations and corresponding co-sets that describe the symmetry of `graph`, given the node and edge equalities given by `node_partitions` and `edge_colors`, respectively.
find_isomorphisms([symmetry])	Find all subgraph isomorphisms between subgraph and graph
is_isomorphic([symmetry])	Returns True if `graph` is isomorphic to `subgraph` and False otherwise.
isomorphisms_iter([symmetry])	Does the same as *find_isomorphisms()* if `graph` and `subgraph` have the same number of nodes.
largest_common_subgraph([symmetry])	Find the largest common induced subgraphs between `subgraph` and `graph`.
subgraph_is_isomorphic([symmetry])	Returns True if a subgraph of `graph` is isomorphic to `subgraph` and False otherwise.
subgraph_isomorphisms_iter([symmetry])	Alternative name for *find_isomorphisms()*.

ISMAGS.analyze_symmetry

ISMAGS.**analyze_symmetry**(*graph*, *node_partitions*, *edge_colors*)

Find a minimal set of permutations and corresponding co-sets that describe the symmetry of `graph`, given the node and edge equalities given by `node_partitions` and `edge_colors`, respectively.

Parameters

graph
[networkx.Graph] The graph whose symmetry should be analyzed.

node_partitions
[list of sets] A list of sets containining node keys. Node keys in the same set are considered equivalent. Every node key in `graph` should be in exactly one of the sets. If all nodes are equivalent, this should be `[set(graph.nodes)]`.

edge_colors
[dict mapping edges to their colors] A dict mapping every edge in `graph` to its corresponding color. Edges with the same color are considered equivalent. If all edges are equivalent, this should be `{e: 0 for e in graph.edges}`.

Returns

set[frozenset]
The found permutations. This is a set of frozensets of pairs of node keys which can be exchanged without changing `subgraph`.

dict[collections.abc.Hashable, set[collections.abc.Hashable]]
The found co-sets. The co-sets is a dictionary of `{node key: set of node keys}`. Every key-value pair describes which `values` can be interchanged without changing nodes less than `key`.

ISMAGS.find_isomorphisms

ISMAGS.**find_isomorphisms**(*symmetry=True*)

Find all subgraph isomorphisms between subgraph and graph

Finds isomorphisms where `subgraph` <= `graph`.

Parameters

symmetry: bool
Whether symmetry should be taken into account. If False, found isomorphisms may be symmetrically equivalent.

Yields

dict
The found isomorphism mappings of {graph_node: subgraph_node}.

ISMAGS.is_isomorphic

ISMAGS.**is_isomorphic**(*symmetry=False*)

Returns True if `graph` is isomorphic to `subgraph` and False otherwise.

Returns

bool

ISMAGS.isomorphisms_iter

ISMAGS.**isomorphisms_iter**(*symmetry=True*)

Does the same as *find_isomorphisms()* if `graph` and `subgraph` have the same number of nodes.

ISMAGS.largest_common_subgraph

ISMAGS.**largest_common_subgraph**(*symmetry=True*)

Find the largest common induced subgraphs between `subgraph` and `graph`.

Parameters

symmetry: bool
Whether symmetry should be taken into account. If False, found largest common subgraphs may be symmetrically equivalent.

Yields

dict
The found isomorphism mappings of {graph_node: subgraph_node}.

ISMAGS.subgraph_is_isomorphic

ISMAGS.**subgraph_is_isomorphic**(*symmetry=False*)

Returns True if a subgraph of `graph` is isomorphic to `subgraph` and False otherwise.

Returns

bool

ISMAGS.subgraph_isomorphisms_iter

ISMAGS.**subgraph_isomorphisms_iter**(*symmetry=True*)

Alternative name for *find_isomorphisms()*.

3.36 Link Analysis

3.36.1 PageRank

PageRank analysis of graph structure.

pagerank(G[, alpha, personalization, ...])	Returns the PageRank of the nodes in the graph.
google_matrix(G[, alpha, personalization, ...])	Returns the Google matrix of the graph.

pagerank

pagerank(*G*, *alpha=0.85*, *personalization=None*, *max_iter=100*, *tol=1e-06*, *nstart=None*, *weight='weight'*, *dangling=None*)

Returns the PageRank of the nodes in the graph.

PageRank computes a ranking of the nodes in the graph G based on the structure of the incoming links. It was originally designed as an algorithm to rank web pages.

Parameters

G
[graph] A NetworkX graph. Undirected graphs will be converted to a directed graph with two directed edges for each undirected edge.

alpha
[float, optional] Damping parameter for PageRank, default=0.85.

personalization: dict, optional
The "personalization vector" consisting of a dictionary with a key some subset of graph nodes and personalization value each of those. At least one personalization value must be non-zero. If not specfiied, a nodes personalization value will be zero. By default, a uniform distribution is used.

max_iter
[integer, optional] Maximum number of iterations in power method eigenvalue solver.

tol
[float, optional] Error tolerance used to check convergence in power method solver.

nstart
[dictionary, optional] Starting value of PageRank iteration for each node.

weight
[key, optional] Edge data key to use as weight. If None weights are set to 1.

dangling: dict, optional
The outedges to be assigned to any "dangling" nodes, i.e., nodes without any outedges. The dict key is the node the outedge points to and the dict value is the weight of that outedge. By default, dangling nodes are given outedges according to the personalization vector (uniform if not specified). This must be selected to result in an irreducible transition matrix (see notes under google_matrix). It may be common to have the dangling dict to be the same as the personalization dict.

Returns

pagerank
[dictionary] Dictionary of nodes with PageRank as value

Raises

PowerIterationFailedConvergence
If the algorithm fails to converge to the specified tolerance within the specified number of iterations of the power iteration method.

See also:

google_matrix

Notes

The eigenvector calculation is done by the power iteration method and has no guarantee of convergence. The iteration will stop after an error tolerance of `len(G) * tol` has been reached. If the number of iterations exceed `max_iter`, a *networkx.exception.PowerIterationFailedConvergence* exception is raised.

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.

References

[1], [2]

Examples

```
>>> G = nx.DiGraph(nx.path_graph(4))
>>> pr = nx.pagerank(G, alpha=0.9)
```

google_matrix

google_matrix(*G*, *alpha=0.85*, *personalization=None*, *nodelist=None*, *weight='weight'*, *dangling=None*)

Returns the Google matrix of the graph.

Parameters

G
[graph] A NetworkX graph. Undirected graphs will be converted to a directed graph with two directed edges for each undirected edge.

alpha
[float] The damping factor.

personalization: dict, optional
The "personalization vector" consisting of a dictionary with a key some subset of graph nodes and personalization value each of those. At least one personalization value must be non-zero. If not specfiied, a nodes personalization value will be zero. By default, a uniform distribution is used.

nodelist
[list, optional] 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
[key, optional] Edge data key to use as weight. If None weights are set to 1.

dangling: dict, optional
The outedges to be assigned to any "dangling" nodes, i.e., nodes without any outedges. The dict key is the node the outedge points to and the dict value is the weight of that outedge. By default, dangling nodes are given outedges according to the personalization vector (uniform if not specified) This must be selected to result in an irreducible transition matrix (see notes below). It may be common to have the dangling dict to be the same as the personalization dict.

Returns

A
[2D NumPy ndarray] Google matrix of the graph

See also:

pagerank

Notes

The array returned represents the transition matrix that describes the Markov chain used in PageRank. For PageRank to converge to a unique solution (i.e., a unique stationary distribution in a Markov chain), the transition matrix must be irreducible. In other words, it must be that there exists a path between every pair of nodes in the graph, or else there is the potential of "rank sinks."

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.

3.36.2 Hits

Hubs and authorities analysis of graph structure.

hits(G[, max_iter, tol, nstart, normalized])	Returns HITS hubs and authorities values for nodes.

hits

hits(*G*, *max_iter=100*, *tol=1e-08*, *nstart=None*, *normalized=True*)

Returns HITS hubs and authorities values for nodes.

The HITS algorithm computes two numbers for a node. Authorities estimates the node value based on the incoming links. Hubs estimates the node value based on outgoing links.

Parameters

G
[graph] A NetworkX graph

max_iter
[integer, optional] Maximum number of iterations in power method.

tol
[float, optional] Error tolerance used to check convergence in power method iteration.

nstart
[dictionary, optional] Starting value of each node for power method iteration.

normalized
[bool (default=True)] Normalize results by the sum of all of the values.

Returns

(hubs,authorities)
[two-tuple of dictionaries] Two dictionaries keyed by node containing the hub and authority values.

Raises

PowerIterationFailedConvergence
If the algorithm fails to converge to the specified tolerance within the specified number of iterations of the power iteration method.

Notes

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.

References

[1], [2]

Examples

```
>>> G = nx.path_graph(4)
>>> h, a = nx.hits(G)
```

3.37 Link Prediction

Link prediction algorithms.

resource_allocation_index(G[, ebunch])	Compute the resource allocation index of all node pairs in ebunch.
jaccard_coefficient(G[, ebunch])	Compute the Jaccard coefficient of all node pairs in ebunch.
adamic_adar_index(G[, ebunch])	Compute the Adamic-Adar index of all node pairs in ebunch.
preferential_attachment(G[, ebunch])	Compute the preferential attachment score of all node pairs in ebunch.
cn_soundarajan_hopcroft(G[, ebunch, community])	Count the number of common neighbors of all node pairs in ebunch
ra_index_soundarajan_hopcroft(G[, ebunch, ...])	Compute the resource allocation index of all node pairs in ebunch using community information.
within_inter_cluster(G[, ebunch, delta, ...])	Compute the ratio of within- and inter-cluster common neighbors of all node pairs in ebunch.
common_neighbor_centrality(G[, ebunch, alpha])	Return the CCPA score for each pair of nodes.

3.37.1 resource_allocation_index

resource_allocation_index(*G*, *ebunch=None*)

Compute the resource allocation index of all node pairs in ebunch.

Resource allocation index of `u` and `v` is defined as

$$\sum_{w \in \Gamma(u) \cap \Gamma(v)} \frac{1}{|\Gamma(w)|}$$

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)] Resource allocation 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 non-existent 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 resource allocation index.

References

[1]

Examples

```
>>> G = nx.complete_graph(5)
>>> preds = nx.resource_allocation_index(G, [(0, 1), (2, 3)])
>>> for u, v, p in preds:
...     print(f"({u}, {v}) -> {p:.8f}")
(0, 1) -> 0.75000000
(2, 3) -> 0.75000000
```

3.37.2 jaccard_coefficient

jaccard_coefficient(*G*, *ebunch=None*)

Compute the Jaccard coefficient of all node pairs in ebunch.

Jaccard coefficient of nodes u and v is defined as

$$\frac{|\Gamma(u) \cap \Gamma(v)|}{|\Gamma(u) \cup \Gamma(v)|}$$

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 non-existent 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.

References

[1]

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
```

3.37.3 adamic_adar_index

adamic_adar_index(*G*, *ebunch=None*)

Compute the Adamic-Adar index of all node pairs in ebunch.

Adamic-Adar index of u and v is defined as

$$\sum_{w \in \Gamma(u) \cap \Gamma(v)} \frac{1}{\log |\Gamma(w)|}$$

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 non-existent 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.

References

[1]

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
```

3.37.4 preferential_attachment

preferential_attachment(*G*, *ebunch=None*)

Compute the preferential attachment score of all node pairs in ebunch.

Preferential attachment score of u and v is defined as

$$|\Gamma(u)||\Gamma(v)|$$

where $\Gamma(u)$ denotes the set of neighbors of u.

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 2-tuples (u, v) where u and v are nodes in the graph. If ebunch is None then all non-existent 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 preferential attachment score.

References

[1]

Examples

```
>>> G = nx.complete_graph(5)
>>> preds = nx.preferential_attachment(G, [(0, 1), (2, 3)])
>>> for u, v, p in preds:
...     print(f"({u}, {v}) -> {p}")
(0, 1) -> 16
(2, 3) -> 16
```

3.37.5 cn_soundarajan_hopcroft

cn_soundarajan_hopcroft(*G*, *ebunch=None*, *community='community'*)

Count the number of common neighbors of all node pairs in ebunch
using community information.

For two nodes u and v, this function computes the number of common neighbors and bonus one for each common neighbor belonging to the same community as u and v. Mathematically,

$$|\Gamma(u) \cap \Gamma(v)| + \sum_{w \in \Gamma(u) \cap \Gamma(v)} f(w)$$

where $f(w)$ equals 1 if w belongs to the same community as u and v or 0 otherwise and $\Gamma(u)$ denotes the set of neighbors of u.

Parameters

G
[graph] A NetworkX undirected graph.

ebunch
[iterable of node pairs, optional (default = None)] The score 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 non-existent edges in the graph will be used. Default value: None.

community
[string, optional (default = 'community')] Nodes attribute name containing the community information. G[u][community] identifies which community u belongs to. Each node belongs to at most one community. Default value: 'community'.

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 score.

References

[1]

Examples

```
>>> G = nx.path_graph(3)
>>> G.nodes[0]["community"] = 0
>>> G.nodes[1]["community"] = 0
>>> G.nodes[2]["community"] = 0
>>> preds = nx.cn_soundarajan_hopcroft(G, [(0, 2)])
>>> for u, v, p in preds:
...     print(f"({u}, {v}) -> {p}")
(0, 2) -> 2
```

3.37.6 ra_index_soundarajan_hopcroft

ra_index_soundarajan_hopcroft(*G*, *ebunch=None*, *community='community'*)

Compute the resource allocation index of all node pairs in ebunch using community information.

For two nodes u and v, this function computes the resource allocation index considering only common neighbors belonging to the same community as u and v. Mathematically,

$$\sum_{w \in \Gamma(u) \cap \Gamma(v)} \frac{f(w)}{|\Gamma(w)|}$$

where $f(w)$ equals 1 if w belongs to the same community as u and v or 0 otherwise and $\Gamma(u)$ denotes the set of neighbors of u.

Parameters

G
[graph] A NetworkX undirected graph.

ebunch
[iterable of node pairs, optional (default = None)] The score 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 non-existent edges in the graph will be used. Default value: None.

community
[string, optional (default = 'community')] Nodes attribute name containing the community information. G[u][community] identifies which community u belongs to. Each node belongs to at most one community. Default value: 'community'.

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 score.

References

[1]

Examples

```
>>> G = nx.Graph()
>>> G.add_edges_from([(0, 1), (0, 2), (1, 3), (2, 3)])
>>> G.nodes[0]["community"] = 0
>>> G.nodes[1]["community"] = 0
>>> G.nodes[2]["community"] = 1
>>> G.nodes[3]["community"] = 0
>>> preds = nx.ra_index_soundarajan_hopcroft(G, [(0, 3)])
>>> for u, v, p in preds:
...     print(f"({u}, {v}) -> {p:.8f}")
(0, 3) -> 0.50000000
```

3.37.7 within_inter_cluster

within_inter_cluster(*G*, *ebunch=None*, *delta=0.001*, *community='community'*)

Compute the ratio of within- and inter-cluster common neighbors of all node pairs in ebunch.

For two nodes `u` and `v`, if a common neighbor `w` belongs to the same community as them, `w` is considered as within-cluster common neighbor of `u` and `v`. Otherwise, it is considered as inter-cluster common neighbor of `u` and `v`. The ratio between the size of the set of within- and inter-cluster common neighbors is defined as the WIC measure. [1]

Parameters

G
[graph] A NetworkX undirected graph.

ebunch
[iterable of node pairs, optional (default = None)] The WIC measure 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 non-existent edges in the graph will be used. Default value: None.

delta
[float, optional (default = 0.001)] Value to prevent division by zero in case there is no inter-cluster common neighbor between two nodes. See [1] for details. Default value: 0.001.

community
[string, optional (default = 'community')] Nodes attribute name containing the community information. G[u][community] identifies which community u belongs to. Each node belongs to at most one community. Default value: 'community'.

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 WIC measure.

References

[1]

Examples

```
>>> G = nx.Graph()
>>> G.add_edges_from([(0, 1), (0, 2), (0, 3), (1, 4), (2, 4), (3, 4)])
>>> G.nodes[0]["community"] = 0
>>> G.nodes[1]["community"] = 1
>>> G.nodes[2]["community"] = 0
>>> G.nodes[3]["community"] = 0
>>> G.nodes[4]["community"] = 0
>>> preds = nx.within_inter_cluster(G, [(0, 4)])
>>> for u, v, p in preds:
...     print(f"({u}, {v}) -> {p:.8f}")
(0, 4) -> 1.99800200
>>> preds = nx.within_inter_cluster(G, [(0, 4)], delta=0.5)
>>> for u, v, p in preds:
...     print(f"({u}, {v}) -> {p:.8f}")
(0, 4) -> 1.33333333
```

3.37.8 common_neighbor_centrality

common_neighbor_centrality(*G*, *ebunch=None*, *alpha=0.8*)

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

$$\alpha \cdot (|\Gamma(u) \cap \Gamma(v)|) + (1 - \alpha) \cdot \frac{N}{d_{uv}}$$

where $\Gamma(u)$ denotes the set of neighbors of u, $\Gamma(v)$ denotes the set of neighbors of v, α 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.

See also:

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 2-tuples (u, v) where u and v are nodes in the graph. If ebunch is None then all non-existent edges in the graph will be used. Default value: None.

alpha
[Parameter defined for participation of Common Neighbor] and Centrality Algorithm share. Values for alpha should normally be between 0 and 1. Default value set to 0.8 because author found better performance at 0.8 for all the dataset. Default value: 0.8

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 Common Neighbor and Centrality based Parameterized Algorithm(CCPA) score.

References

[1]

Examples

```
>>> G = nx.complete_graph(5)
>>> preds = nx.common_neighbor_centrality(G, [(0, 1), (2, 3)])
>>> for u, v, p in preds:
...     print(f"({u}, {v}) -> {p}")
(0, 1) -> 3.4000000000000004
(2, 3) -> 3.4000000000000004
```

3.38 Lowest Common Ancestor

Algorithms for finding the lowest common ancestor of trees and DAGs.

all_pairs_lowest_common_ancestor(G[, pairs])	Return the lowest common ancestor of all pairs or the provided pairs
tree_all_pairs_lowest_common_ancestor(G[, ...])	Yield the lowest common ancestor for sets of pairs in a tree.
lowest_common_ancestor(G, node1, node2[, ...])	Compute the lowest common ancestor of the given pair of nodes.

3.38.1 all_pairs_lowest_common_ancestor

all_pairs_lowest_common_ancestor(*G*, *pairs=None*)

Return the lowest common ancestor of all pairs or the provided pairs

Parameters

G
[NetworkX directed graph]

pairs
[iterable of pairs of nodes, optional (default: all pairs)] The pairs of nodes of interest. If None, will find the LCA of all pairs of nodes.

Yields

((node1, node2), lca)
[2-tuple] Where lca is least common ancestor of node1 and node2. Note that for the default case, the order of the node pair is not considered, e.g. you will not get both `(a, b)` and `(b, a)`

Raises

NetworkXPointlessConcept
If `G` is null.

NetworkXError
If `G` is not a DAG.

See also:

lowest_common_ancestor

Notes

Only defined on non-null directed acyclic graphs.

Examples

The default behavior is to yield the lowest common ancestor for all possible combinations of nodes in `G`, including self-pairings:

```
>>> G = nx.DiGraph([(0, 1), (0, 3), (1, 2)])
>>> dict(nx.all_pairs_lowest_common_ancestor(G))
{(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}
```

The pairs argument can be used to limit the output to only the specified node pairings:

```
>>> dict(nx.all_pairs_lowest_common_ancestor(G, pairs=[(1, 2), (2, 3)]))
{(1, 2): 1, (2, 3): 0}
```

3.38.2 tree_all_pairs_lowest_common_ancestor

tree_all_pairs_lowest_common_ancestor(*G*, *root=None*, *pairs=None*)

Yield the lowest common ancestor for sets of pairs in a tree.

Parameters

G
[NetworkX directed graph (must be a tree)]

root
[node, optional (default: None)] The root of the subtree to operate on. If None, assume the entire graph has exactly one source and use that.

pairs
[iterable or iterator of pairs of nodes, optional (default: None)] The pairs of interest. If None, Defaults to all pairs of nodes under `root` that have a lowest common ancestor.

Returns

lcas
[generator of tuples `((u, v), lca)` where `u` and `v` are nodes] in `pairs` and `lca` is their lowest common ancestor.

See also:

all_pairs_lowest_common_ancestor
similar routine for general DAGs

lowest_common_ancestor
just a single pair for general DAGs

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.

Examples

```
>>> 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}
```

3.38.3 lowest_common_ancestor

lowest_common_ancestor(*G*, *node1*, *node2*, *default=None*)

Compute the lowest common ancestor of the given pair of nodes.

Parameters

G
[NetworkX directed graph]

node1, node2
[nodes in the graph.]

default
[object] Returned if no common ancestor between `node1` and `node2`

Returns

**The lowest common ancestor of node1 and node2,
or default if they have no common ancestors.**

See also:

all_pairs_lowest_common_ancestor

Examples

```
>>> G = nx.DiGraph()
>>> nx.add_path(G, (0, 1, 2, 3))
>>> nx.add_path(G, (0, 4, 3))
>>> nx.lowest_common_ancestor(G, 2, 4)
0
```

3.39 Matching

Functions for computing and verifying matchings in a graph.

is_matching(G, matching)	Return True if `matching` is a valid matching of `G`
is_maximal_matching(G, matching)	Return True if `matching` is a maximal matching of `G`
is_perfect_matching(G, matching)	Return True if `matching` is a perfect matching for `G`
maximal_matching(G)	Find a maximal matching in the graph.
max_weight_matching(G[, maxcardinality, weight])	Compute a maximum-weighted matching of G.
min_weight_matching(G[, weight])	Computing a minimum-weight maximal matching of G.

3.39.1 is_matching

is_matching(*G*, *matching*)

Return True if `matching` is a valid matching of `G`

A *matching* in a graph is a set of edges in which no two distinct edges share a common endpoint. Each node is incident to at most one edge in the matching. The edges are said to be independent.

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 matching in the graph.

Raises

NetworkXError
If the proposed matching has an edge to a node not in G. Or if the matching is not a collection of 2-tuple edges.

Examples

```
>>> G = nx.Graph([(1, 2), (1, 3), (2, 3), (2, 4), (3, 5), (4, 5)])
>>> nx.is_maximal_matching(G, {1: 3, 2: 4})  # using dict to represent matching
True
```

```
>>> nx.is_matching(G, {(1, 3), (2, 4)})  # using set to represent matching
True
```

3.39.2 is_maximal_matching

is_maximal_matching(*G*, *matching*)

Return True if `matching` is a maximal matching of `G`

A *maximal matching* in a graph is a matching in which adding any edge would cause the set to no longer be a valid matching.

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
```

3.39.3 is_perfect_matching

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
```

3.39.4 maximal_matching

maximal_matching(*G*)

Find a maximal matching in the graph.

A matching is a subset of edges in which no node occurs more than once. A maximal matching cannot add more edges and still be a matching.

Parameters

G
[NetworkX graph] Undirected graph

Returns

matching
[set] A maximal matching of the graph.

Notes

The algorithm greedily selects a maximal matching M of the graph G (i.e. no superset of M exists). It runs in $O(|E|)$ time.

Examples

```
>>> G = nx.Graph([(1, 2), (1, 3), (2, 3), (2, 4), (3, 5), (4, 5)])
>>> sorted(nx.maximal_matching(G))
[(1, 2), (3, 5)]
```

3.39.5 max_weight_matching

max_weight_matching(*G*, *maxcardinality=False*, *weight='weight'*)

Compute a maximum-weighted matching of G.

A matching is a subset of edges in which no node occurs more than once. The weight of a matching is the sum of the weights of its edges. A maximal matching cannot add more edges and still be a matching. The cardinality of a matching is the number of matched edges.

Parameters

G
[NetworkX graph] Undirected graph

maxcardinality: bool, optional (default=False)
If maxcardinality is True, compute the maximum-cardinality matching with maximum weight among all maximum-cardinality matchings.

weight: string, optional (default='weight')
Edge data key corresponding to the edge weight. If key not found, uses 1 as weight.

Returns

matching
[set] A maximal matching of the graph.

Notes

If G has edges with weight attributes the edge data are used as weight values else the weights are assumed to be 1.

This function takes time O(number_of_nodes ** 3).

If all edge weights are integers, the algorithm uses only integer computations. If floating point weights are used, the algorithm could return a slightly suboptimal matching due to numeric precision errors.

This method is based on the "blossom" method for finding augmenting paths and the "primal-dual" method for finding a matching of maximum weight, both methods invented by Jack Edmonds [1].

Bipartite graphs can also be matched using the functions present in *networkx.algorithms.bipartite.matching*.

References

[1]

Examples

```
>>> G = nx.Graph()
>>> edges = [(1, 2, 6), (1, 3, 2), (2, 3, 1), (2, 4, 7), (3, 5, 9), (4, 5, 3)]
>>> G.add_weighted_edges_from(edges)
>>> sorted(nx.max_weight_matching(G))
[(2, 4), (5, 3)]
```

3.39.6 min_weight_matching

min_weight_matching(*G*, *weight='weight'*)

Computing a minimum-weight maximal matching of G.

Use the maximum-weight algorithm with edge weights subtracted from the maximum weight of all edges.

A matching is a subset of edges in which no node occurs more than once. The weight of a matching is the sum of the weights of its edges. A maximal matching cannot add more edges and still be a matching. The cardinality of a matching is the number of matched edges.

This method replaces the edge weights with 1 plus the maximum edge weight minus the original edge weight.

new_weight = (max_weight + 1) - edge_weight

then runs *max_weight_matching()* with the new weights. The max weight matching with these new weights corresponds to the min weight matching using the original weights. Adding 1 to the max edge weight keeps all edge weights positive and as integers if they started as integers.

You might worry that adding 1 to each weight would make the algorithm favor matchings with more edges. But we use the parameter `maxcardinality=True` in *max_weight_matching* to ensure that the number of edges in the competing matchings are the same and thus the optimum does not change due to changes in the number of edges.

Read the documentation of *max_weight_matching* for more information.

Parameters

G
[NetworkX graph] Undirected graph

weight: string, optional (default='weight')
Edge data key corresponding to the edge weight. If key not found, uses 1 as weight.

Returns

matching
[set] A minimal weight matching of the graph.

See also:

max_weight_matching

3.40 Minors

Subpackages related to graph-minor problems.

In graph theory, an undirected graph H is called a minor of the graph G if H can be formed from G by deleting edges and vertices and by contracting edges [1].

3.40.1 References

contracted_edge(G, edge[, self_loops, copy])	Returns the graph that results from contracting the specified edge.
contracted_nodes(G, u, v[, self_loops, copy])	Returns the graph that results from contracting `u` and `v`.
identified_nodes(G, u, v[, self_loops, copy])	Returns the graph that results from contracting `u` and `v`.
equivalence_classes(iterable, relation)	Returns equivalence classes of `relation` when applied to `iterable`.
quotient_graph(G, partition[, ...])	Returns the quotient graph of `G` under the specified equivalence relation on nodes.

3.40.2 contracted_edge

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 graph.

Parameters

G
[NetworkX graph] The graph whose edge will be contracted.

edge
[tuple] Must be a pair of nodes in `G`.

self_loops
[Boolean] If this is True, any edges (including `edge`) joining the endpoints of `edge` in `G` become self-loops on the new node in the returned graph.

copy
[Boolean (default True)] If this is True, a the contraction will be performed on a copy of `G`, otherwise the contraction will happen in place.

Returns

Networkx graph
A new graph object of the same type as `G` (leaving `G` unmodified) with endpoints of `edge` identified in a single node. The right node of `edge` will be merged into the left one, so only the left one will appear in the returned graph.

Raises

ValueError
If `edge` is not an edge in `G`.

See also:

contracted_nodes
quotient_graph

Examples

Attempting to contract two nonadjacent nodes yields an error:

```
>>> G = nx.cycle_graph(4)
>>> nx.contracted_edge(G, (1, 3))
Traceback (most recent call last):
  ...
ValueError: Edge (1, 3) does not exist in graph G; cannot contract it
```

Contracting two adjacent nodes in the cycle graph on *n* nodes yields the cycle graph on *n - 1* nodes:

```
>>> C5 = nx.cycle_graph(5)
>>> C4 = nx.cycle_graph(4)
>>> M = nx.contracted_edge(C5, (0, 1), self_loops=False)
>>> nx.is_isomorphic(M, C4)
True
```

3.40.3 contracted_nodes

contracted_nodes(*G*, *u*, *v*, *self_loops=True*, *copy=True*)

Returns the graph that results from contracting `u` and `v`.

Node contraction identifies the two nodes as a single node incident to any edge that was incident to the original two nodes.

Parameters

G
[NetworkX graph] The graph whose nodes will be contracted.

u, v
[nodes] Must be nodes in `G`.

self_loops
[Boolean] If this is True, any edges joining `u` and `v` in `G` become self-loops on the new node in the returned graph.

copy
[Boolean] If this is True (default True), make a copy of `G` and return that instead of directly changing `G`.

Returns

Networkx graph
If Copy is True, A new graph object of the same type as `G` (leaving `G` unmodified) with `u` and `v` identified in a single node. The right node `v` will be merged into the node `u`, so only `u` will appear in the returned graph. If copy is False, Modifies `G` with `u` and `v` identified in a single node. The right node `v` will be merged into the node `u`, so only `u` will appear in the returned graph.

See also:

contracted_edge
quotient_graph

Notes

For multigraphs, the edge keys for the realigned edges may not be the same as the edge keys for the old edges. This is natural because edge keys are unique only within each pair of nodes.

For non-multigraphs where `u` and `v` are adjacent to a third node `w`, the edge (`v`, `w`) will be contracted into the edge (`u`, `w`) with its attributes stored into a "contraction" attribute.

This function is also available as *identified_nodes*.

Examples

Contracting two nonadjacent nodes of the cycle graph on four nodes `C_4` yields the path graph (ignoring parallel edges):

```
>>> 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)]
```

3.40.4 identified_nodes

identified_nodes(*G*, *u*, *v*, *self_loops=True*, *copy=True*)

Returns the graph that results from contracting `u` and `v`.

Node contraction identifies the two nodes as a single node incident to any edge that was incident to the original two nodes.

Parameters

G
: [NetworkX graph] The graph whose nodes will be contracted.

u, v
: [nodes] Must be nodes in `G`.

self_loops
: [Boolean] If this is True, any edges joining `u` and `v` in `G` become self-loops on the new node in the returned graph.

copy
: [Boolean] If this is True (default True), make a copy of `G` and return that instead of directly changing `G`.

Returns

Networkx graph
: If Copy is True, A new graph object of the same type as `G` (leaving `G` unmodified) with `u` and `v` identified in a single node. The right node `v` will be merged into the node `u`, so only `u` will appear in the returned graph. If copy is False, Modifies `G` with `u` and `v` identified in a single node. The right node `v` will be merged into the node `u`, so only `u` will appear in the returned graph.

See also:

contracted_edge
quotient_graph

Notes

For multigraphs, the edge keys for the realigned edges may not be the same as the edge keys for the old edges. This is natural because edge keys are unique only within each pair of nodes.

For non-multigraphs where `u` and `v` are adjacent to a third node `w`, the edge (`v`, `w`) will be contracted into the edge (`u`, `w`) with its attributes stored into a "contraction" attribute.

This function is also available as *identified_nodes*.

Examples

Contracting two nonadjacent nodes of the cycle graph on four nodes `C_4` yields the path graph (ignoring parallel edges):

```
>>> 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)]
```

3.40.5 equivalence_classes

equivalence_classes(*iterable*, *relation*)

Returns equivalence classes of `relation` when applied to `iterable`.

The equivalence classes, or blocks, consist of objects from `iterable` which are all equivalent. They are defined to be equivalent if the `relation` function returns `True` when passed any two objects from that class, and `False` otherwise. To define an equivalence relation the function must be reflexive, symmetric and transitive.

Parameters

iterable
[list, tuple, or set] An iterable of elements/nodes.

relation
[function] A Boolean-valued function that implements an equivalence relation (reflexive, symmetric, transitive binary relation) on the elements of `iterable` - it must take two elements and return `True` if they are related, or `False` if not.

Returns

set of frozensets
A set of frozensets representing the partition induced by the equivalence 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 `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)
{frozenset({1, 4, 7}), frozenset({8, 2, 5}), frozenset({0, 9, 3, 6})}
```

3.40.6 quotient_graph

quotient_graph(*G*, *partition*, *edge_relation=None*, *node_data=None*, *edge_data=None*, *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`.

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.

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.

relabel
[bool] If True, relabel the nodes of the quotient graph to be nonnegative integers. Otherwise, the nodes are identified with `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 `set` instances and `relabel` is False, each node will be a `frozenset` corresponding to the same `set`.

Raises

NetworkXException
If the given partition is not a valid partition of the nodes of `G`.

References

[1]

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 `condensation()`). This example comes from the Wikipedia article *`Strongly connected component`_*.

```
>>> G = nx.DiGraph()
>>> edges = [
...     "ab",
...     "be",
...     "bf",
...     "bc",
...     "cg",
...     "cd",
...     "dc",
...     "dh",
...     "ea",
...     "ef",
...     "fg",
...     "gf",
...     "hd",
...     "hf",
... ]
>>> G.add_edges_from(tuple(x) for x in edges)
>>> components = list(nx.strongly_connected_components(G))
>>> sorted(sorted(component) for component in components)
[['a', 'b', 'e'], ['c', 'd', 'h'], ['f', 'g']]
>>>
>>> C = nx.condensation(G, components)
>>> component_of = C.graph["mapping"]
>>> same_component = lambda u, v: component_of[u] == component_of[v]
>>> Q = nx.quotient_graph(G, same_component)
>>> nx.is_isomorphic(C, Q)
True
```

Node identification can be represented as the quotient of a graph under the equivalence relation that places the two nodes in one block and each other node in its own singleton block.

```
>>> K24 = nx.complete_bipartite_graph(2, 4)
>>> K34 = nx.complete_bipartite_graph(3, 4)
```

(continues on next page)

(continued from previous page)

```
>>> C = nx.contracted_nodes(K34, 1, 2)
>>> nodes = {1, 2}
>>> is_contracted = lambda u, v: u in nodes and v in nodes
>>> Q = nx.quotient_graph(K34, is_contracted)
>>> nx.is_isomorphic(Q, C)
True
>>> nx.is_isomorphic(Q, K24)
True
```

The blockmodeling technique described in [1] can be implemented as a quotient graph.

```
>>> G = nx.path_graph(6)
>>> partition = [{0, 1}, {2, 3}, {4, 5}]
>>> M = nx.quotient_graph(G, partition, relabel=True)
>>> list(M.edges())
[(0, 1), (1, 2)]
```

Here is the sample example but using partition as a dict of block sets.

```
>>> G = nx.path_graph(6)
>>> partition = {0: {0, 1}, 2: {2, 3}, 4: {4, 5}}
>>> M = nx.quotient_graph(G, partition, relabel=True)
>>> list(M.edges())
[(0, 1), (1, 2)]
```

Partitions can be represented in various ways:

0. a list/tuple/set of block lists/tuples/sets
1. a dict with block labels as keys and blocks lists/tuples/sets as values
2. a dict with block lists/tuples/sets as keys and block labels as values
3. a function from nodes in the original iterable to block labels
4. an equivalence relation function on the target iterable

As *quotient_graph* is designed to accept partitions represented as (0), (1) or (4) only, the *equivalence_classes* function can be used to get the partitions in the right form, in order to call *quotient_graph*.

3.41 Maximal independent set

Algorithm to find a maximal (not maximum) independent set.

maximal_independent_set(G[, nodes, seed])	Returns a random maximal independent set guaranteed to contain a given set of nodes.

3.41.1 maximal_independent_set

maximal_independent_set(*G*, *nodes=None*, *seed=None*)

Returns a random maximal independent set guaranteed to contain a given set of nodes.

An independent set is a set of nodes such that the subgraph of G induced by these nodes contains no edges. A maximal independent set is an independent set such that it is not possible to add a new node and still get an independent set.

Parameters

G
[NetworkX graph]

nodes
[list or iterable] Nodes that must be part of the independent set. This set of nodes must be independent.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

indep_nodes
[list] List of nodes that are part of a maximal independent set.

Raises

NetworkXUnfeasible
If the nodes in the provided list are not part of the graph or do not form an independent set, an exception is raised.

NetworkXNotImplemented
If `G` is directed.

Notes

This algorithm does not solve the maximum independent set problem.

Examples

```
>>> G = nx.path_graph(5)
>>> nx.maximal_independent_set(G)
[4, 0, 2]
>>> nx.maximal_independent_set(G, [1])
[1, 3]
```

3.42 non-randomness

Computation of graph non-randomness

non_randomness(G[, k, weight])	Compute the non-randomness of graph G.

3.42.1 non_randomness

non_randomness(*G*, *k=None*, *weight='weight'*)

Compute the non-randomness of graph G.

The first returned value nr is the sum of non-randomness values of all edges within the graph (where the non-randomness of an edge tends to be small when the two nodes linked by that edge are from two different communities).

The second computed value nr_rd is a relative measure that indicates to what extent graph G is different from random graphs in terms of probability. When it is close to 0, the graph tends to be more likely generated by an Erdos Renyi model.

Parameters

G
[NetworkX graph] Graph must be symmetric, connected, and without self-loops.

k
[int] The number of communities in G. If k is not set, the function will use a default community detection algorithm to set it.

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, i.e., the graph is binary.

Returns

non-randomness
[(float, float) tuple] Non-randomness, Relative non-randomness w.r.t. Erdos Renyi random graphs.

Raises

NetworkXException
if the input graph is not connected.

NetworkXError
if the input graph contains self-loops.

Notes

This computes Eq. (4.4) and (4.5) in Ref. [1].

If a weight field is passed, this algorithm will use the eigenvalues of the weighted adjacency matrix to compute Eq. (4.4) and (4.5).

References

[1]

Examples

```
>>> G = nx.karate_club_graph()
>>> nr, nr_rd = nx.non_randomness(G, 2)
>>> nr, nr_rd = nx.non_randomness(G, 2, 'weight')
```

3.43 Moral

Function for computing the moral graph of a directed graph.

moral_graph(G)	Return the Moral Graph

3.43.1 moral_graph

moral_graph(*G*)

Return the Moral Graph

Returns the moralized graph of a given directed graph.

Parameters

G
[NetworkX graph] Directed graph

Returns

H
[NetworkX graph] The undirected moralized graph of G

Raises

NetworkXNotImplemented
If G is undirected.

Notes

A moral graph is an undirected graph H = (V, E) generated from a directed Graph, where if a node has more than one parent node, edges between these parent nodes are inserted and all directed edges become undirected.

https://en.wikipedia.org/wiki/Moral_graph

References

[1]

Examples

```
>>> G = nx.DiGraph([(1, 2), (2, 3), (2, 5), (3, 4), (4, 3)])
>>> G_moral = nx.moral_graph(G)
>>> G_moral.edges()
EdgeView([(1, 2), (2, 3), (2, 5), (2, 4), (3, 4)])
```

3.44 Node Classification

This module provides the functions for node classification problem.

The functions in this module are not imported into the top level `networkx` namespace. You can access these functions by importing the *networkx.algorithms.node_classification* modules, then accessing the functions as attributes of `node_classification`. For example:

```
>>> from networkx.algorithms import node_classification
>>> G = nx.path_graph(4)
>>> G.edges()
EdgeView([(0, 1), (1, 2), (2, 3)])
>>> G.nodes[0]["label"] = "A"
>>> G.nodes[3]["label"] = "B"
>>> node_classification.harmonic_function(G)
['A', 'A', 'B', 'B']
```

3.44.1 References

Zhu, X., Ghahramani, Z., & Lafferty, J. (2003, August). Semi-supervised learning using gaussian fields and harmonic functions. In ICML (Vol. 3, pp. 912-919).

harmonic_function(G[, max_iter, label_name])	Node classification by Harmonic function
local_and_global_consistency(G[, alpha, ...])	Node classification by Local and Global Consistency

3.44.2 harmonic_function

harmonic_function(*G*, *max_iter=30*, *label_name='label'*)

Node classification by Harmonic function

Function for computing Harmonic function algorithm by Zhu et al.

Parameters

G
[NetworkX Graph]

max_iter
[int] maximum number of iterations allowed

label_name
[string] name of target labels to predict

Returns

predicted
[list] List of length `len(G)` with the predicted labels for each node.

Raises

NetworkXError
If no nodes in `G` have attribute `label_name`.

References

Zhu, X., Ghahramani, Z., & Lafferty, J. (2003, August). Semi-supervised learning using gaussian fields and harmonic functions. In ICML (Vol. 3, pp. 912-919).

Examples

```
>>> from networkx.algorithms import node_classification
>>> G = nx.path_graph(4)
>>> G.nodes[0]["label"] = "A"
>>> G.nodes[3]["label"] = "B"
>>> G.nodes(data=True)
NodeDataView({0: {'label': 'A'}, 1: {}, 2: {}, 3: {'label': 'B'}})
>>> G.edges()
EdgeView([(0, 1), (1, 2), (2, 3)])
>>> predicted = node_classification.harmonic_function(G)
>>> predicted
['A', 'A', 'B', 'B']
```

3.44.3 local_and_global_consistency

local_and_global_consistency(*G*, *alpha=0.99*, *max_iter=30*, *label_name='label'*)

Node classification by Local and Global Consistency

Function for computing Local and global consistency algorithm by Zhou et al.

Parameters

G
[NetworkX Graph]

alpha
[float] Clamping factor

max_iter
[int] Maximum number of iterations allowed

label_name
[string] Name of target labels to predict

Returns

predicted
[list] List of length `len(G)` with the predicted labels for each node.

Raises

NetworkXError
If no nodes in `G` have attribute `label_name`.

References

Zhou, D., Bousquet, O., Lal, T. N., Weston, J., & Schölkopf, B. (2004). Learning with local and global consistency. Advances in neural information processing systems, 16(16), 321-328.

Examples

```
>>> from networkx.algorithms import node_classification
>>> G = nx.path_graph(4)
>>> G.nodes[0]["label"] = "A"
>>> G.nodes[3]["label"] = "B"
>>> G.nodes(data=True)
NodeDataView({0: {'label': 'A'}, 1: {}, 2: {}, 3: {'label': 'B'}})
>>> G.edges()
EdgeView([(0, 1), (1, 2), (2, 3)])
>>> predicted = node_classification.local_and_global_consistency(G)
>>> predicted
['A', 'A', 'B', 'B']
```

3.45 Operators

Unary operations on graphs

complement(G)	Returns the graph complement of G.
reverse(G[, copy])	Returns the reverse directed graph of G.

3.45.1 complement

complement(*G*)

Returns the graph complement of G.

Parameters

G
[graph] A NetworkX graph

Returns

GC
[A new graph.]

Notes

Note that *complement* does not create self-loops and also does not produce parallel edges for MultiGraphs.

Graph, node, and edge data are not propagated to the new graph.

Examples

```
>>> G = nx.Graph([(1, 2), (1, 3), (2, 3), (3, 4), (3, 5)])
>>> G_complement = nx.complement(G)
>>> G_complement.edges() # This shows the edges of the complemented graph
EdgeView([(1, 4), (1, 5), (2, 4), (2, 5), (4, 5)])
```

3.45.2 reverse

reverse(*G*, *copy=True*)

Returns the reverse directed graph of G.

Parameters

G
[directed graph] A NetworkX directed graph

copy
[bool] If True, then a new graph is returned. If False, then the graph is reversed in place.

Returns

H
[directed graph] The reversed G.

Raises

NetworkXError
If graph is undirected.

Examples

```
>>> G = nx.DiGraph([(1, 2), (1, 3), (2, 3), (3, 4), (3, 5)])
>>> G_reversed = nx.reverse(G)
>>> G_reversed.edges()
OutEdgeView([(2, 1), (3, 1), (3, 2), (4, 3), (5, 3)])
```

Operations on graphs including union, intersection, difference.

compose(G, H)	Compose graph G with H by combining nodes and edges into a single graph.
union(G, H[, rename])	Combine graphs G and H.
disjoint_union(G, H)	Combine graphs G and H.
intersection(G, H)	Returns a new graph that contains only the nodes and the edges that exist in both G and H.
difference(G, H)	Returns a new graph that contains the edges that exist in G but not in H.
symmetric_difference(G, H)	Returns new graph with edges that exist in either G or H but not both.
full_join(G, H[, rename])	Returns the full join of graphs G and H.

3.45.3 compose

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:

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)
>>> R.nodes
NodeView((0, 1, 2))
>>> R.edges
EdgeView([(0, 1), (0, 2), (1, 2)])
```

By default, the attributes from `H` take precedent over attributes from `G`. If you prefer another way of combining attributes, you can update them after the compose operation:

```
>>> G = nx.Graph([(0, 1, {'weight': 2.0}), (3, 0, {'weight': 100.0})])
>>> H = nx.Graph([(0, 1, {'weight': 10.0}), (1, 2, {'weight': -1.0})])
>>> nx.set_node_attributes(G, {0: 'dark', 1: 'light', 3: 'black'}, name='color')
>>> nx.set_node_attributes(H, {0: 'green', 1: 'orange', 2: 'yellow'}, name='color
↪')
>>> GcomposeH = nx.compose(G, H)
```

Normally, color attribute values of nodes of GcomposeH come from H. We can workaround this as follows:

```
>>> node_data = {n: G.nodes[n]['color'] + " " + H.nodes[n]['color'] for n in G.
↪nodes & H.nodes}
>>> nx.set_node_attributes(GcomposeH, node_data, 'color')
>>> print(GcomposeH.nodes[0]['color'])
dark green
```

```
>>> print(GcomposeH.nodes[3]['color'])
black
```

Similarly, we can update edge attributes after the compose operation in a way we prefer:

```
>>> edge_data = {e: G.edges[e]['weight'] * H.edges[e]['weight'] for e in G.edges &
↪ H.edges}
>>> nx.set_edge_attributes(GcomposeH, edge_data, 'weight')
>>> print(GcomposeH.edges[(0, 1)]['weight'])
20.0
```

```
>>> print(GcomposeH.edges[(3, 0)]['weight'])
100.0
```

3.45.4 union

union(*G*, *H*, *rename=()*)

Combine graphs G and H. The names of nodes must be unique.

A name collision between the graphs will raise an exception.

A renaming facility is provided to avoid name collisions.

Parameters

G, H
[graph] A NetworkX graph

rename
[iterable , optional] Node names of G and H can be changed by specifying the tuple rename=('G-','H-') (for example). Node "u" in G is then renamed "G-u" and "v" in H is renamed "H-v".

Returns

U
[A union graph with the same type as G.]

See also:

compose
update()
disjoint_union

Notes

To combine graphs that have common nodes, consider compose(G, H) or the method, Graph.update().

disjoint_union() is similar to union() except that it avoids name clashes by relabeling the nodes with sequential integers.

Edge and node attributes are propagated from G and H to the union graph. Graph attributes are also propagated, but if they are present in both G and H, then the value from H is used.

Examples

```
>>> G = nx.Graph([(0, 1), (0, 2), (1, 2)])
>>> H = nx.Graph([(0, 1), (0, 3), (1, 3), (1, 2)])
>>> U = nx.union(G, H, rename=("G", "H"))
>>> U.nodes
NodeView(('G0', 'G1', 'G2', 'H0', 'H1', 'H3', 'H2'))
>>> U.edges
EdgeView([('G0', 'G1'), ('G0', 'G2'), ('G1', 'G2'), ('H0', 'H1'), ('H0', 'H3'), (
↪'H1', 'H3'), ('H1', 'H2')])
```

3.45.5 disjoint_union

disjoint_union(*G*, *H*)

Combine graphs G and H. The nodes are assumed to be unique (disjoint).

This algorithm automatically relabels nodes to avoid name collisions.

Parameters

G,H
[graph] A NetworkX graph

Returns

U
[A union graph with the same type as G.]

See also:

union
compose
update()

Notes

A new graph is created, of the same class as G. It is recommended that G and H be either both directed or both undirected.

The nodes of G are relabeled 0 to len(G)-1, and the nodes of H are relabeled len(G) to len(G)+len(H)-1.

Renumbering forces G and H to be disjoint, so no exception is ever raised for a name collision. To preserve the check for common nodes, use union().

Edge and node attributes are propagated from G and H to the union graph. Graph attributes are also propagated, but if they are present in both G and H, then the value from H is used.

To combine graphs that have common nodes, consider compose(G, H) or the method, Graph.update().

Examples

```
>>> G = nx.Graph([(0, 1), (0, 2), (1, 2)])
>>> H = nx.Graph([(0, 3), (1, 2), (2, 3)])
>>> G.nodes[0]["key1"] = 5
>>> H.nodes[0]["key2"] = 10
>>> U = nx.disjoint_union(G, H)
>>> U.nodes(data=True)
NodeDataView({0: {'key1': 5}, 1: {}, 2: {}, 3: {'key2': 10}, 4: {}, 5: {}, 6: {}})
>>> U.edges
EdgeView([(0, 1), (0, 2), (1, 2), (3, 4), (4, 6), (5, 6)])
```

3.45.6 intersection

intersection(*G*, *H*)

Returns a new graph that contains only the nodes and the edges that exist in both G and H.

Parameters

G,H
[graph] A NetworkX graph. G and H can have different node sets but must be both graphs or both multigraphs.

Returns

GH
[A new graph with the same type as G.]

Raises

NetworkXError
If one is a MultiGraph and the other one is a graph.

Notes

Attributes from the graph, nodes, and edges are not copied to the new graph. If you want a new graph of the intersection of G and H with the attributes (including edge data) from G use remove_nodes_from() as follows

```
>>> G = nx.path_graph(3)
>>> H = nx.path_graph(5)
>>> R = G.copy()
>>> R.remove_nodes_from(n for n in G if n not in H)
>>> R.remove_edges_from(e for e in G.edges if e not in H.edges)
```

Examples

```
>>> G = nx.Graph([(0, 1), (0, 2), (1, 2)])
>>> H = nx.Graph([(0, 3), (1, 2), (2, 3)])
>>> R = nx.intersection(G, H)
>>> R.nodes
NodeView((0, 1, 2))
>>> R.edges
EdgeView([(1, 2)])
```

3.45.7 difference

difference(*G*, *H*)

Returns a new graph that contains the edges that exist in G but not in H.

The node sets of H and G must be the same.

Parameters

G,H
[graph] A NetworkX graph. G and H must have the same node sets.

Returns

D
[A new graph with the same type as G.]

Notes

Attributes from the graph, nodes, and edges are not copied to the new graph. If you want a new graph of the difference of G and H with the attributes (including edge data) from G use remove_nodes_from() as follows:

```
>>> G = nx.path_graph(3)
>>> H = nx.path_graph(5)
>>> R = G.copy()
>>> R.remove_nodes_from(n for n in G if n in H)
```

Examples

```
>>> G = nx.Graph([(0, 1), (0, 2), (1, 2), (1, 3)])
>>> H = nx.Graph([(0, 1), (1, 2), (0, 3)])
>>> R = nx.difference(G, H)
>>> R.nodes
NodeView((0, 1, 2, 3))
>>> R.edges
EdgeView([(0, 2), (1, 3)])
```

3.45.8 symmetric_difference

symmetric_difference(*G*, *H*)

Returns new graph with edges that exist in either G or H but not both.

The node sets of H and G must be the same.

Parameters

G,H
[graph] A NetworkX graph. G and H must have the same node sets.

Returns

D
[A new graph with the same type as G.]

Notes

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)])
```

3.45.9 full_join

full_join(*G*, *H*, *rename=(None, None)*)

Returns the full join of graphs G and H.

Full join is the union of G and H in which all edges between G and H are added. The node sets of G and H must be disjoint, otherwise an exception is raised.

Parameters

G, H
[graph] A NetworkX graph

rename
[tuple , default=(None, None)] Node names of G and H can be changed by specifying the tuple rename=('G-','H-') (for example). Node "u" in G is then renamed "G-u" and "v" in H is renamed "H-v".

Returns

U
[The full join graph with the same type as G.]

See also:

union
disjoint_union

Notes

It is recommended that G and H be either both directed or both undirected.

If G is directed, then edges from G to H are added as well as from H to G.

Note that full_join() does not produce parallel edges for MultiGraphs.

The full join operation of graphs G and H is the same as getting their complement, performing a disjoint union, and finally getting the complement of the resulting graph.

Graph, edge, and node attributes are propagated from G and H to the union graph. If a graph attribute is present in both G and H the value from H is used.

Examples

```
>>> G = nx.Graph([(0, 1), (0, 2)])
>>> H = nx.Graph([(3, 4)])
>>> R = nx.full_join(G, H, rename=("G", "H"))
>>> R.nodes
NodeView(('G0', 'G1', 'G2', 'H3', 'H4'))
>>> R.edges
EdgeView([('G0', 'G1'), ('G0', 'G2'), ('G0', 'H3'), ('G0', 'H4'), ('G1', 'H3'), (
↪'G1', 'H4'), ('G2', 'H3'), ('G2', 'H4'), ('H3', 'H4')])
```

Operations on many graphs.

compose_all(graphs)	Returns the composition of all graphs.
union_all(graphs[, rename])	Returns the union of all graphs.
disjoint_union_all(graphs)	Returns the disjoint union of all graphs.
intersection_all(graphs)	Returns a new graph that contains only the nodes and the edges that exist in all graphs.

3.45.10 compose_all

compose_all(*graphs*)

Returns the composition of all graphs.

Composition is the simple union of the node sets and edge sets. The node sets of the supplied graphs need not be disjoint.

Parameters

graphs
[iterable] Iterable of NetworkX graphs

Returns

C
[A graph with the same type as the first graph in list]

Raises

ValueError
If `graphs` is an empty list.

Notes

It is recommended that the supplied graphs be either all directed or all undirected.

Graph, edge, and node attributes are propagated to the union graph. If a graph attribute is present in multiple graphs, then the value from the last graph in the list with that attribute is used.

3.45.11 union_all

union_all(*graphs*, *rename=()*)

Returns the union of all graphs.

The graphs must be disjoint, otherwise an exception is raised.

Parameters

graphs
[iterable] Iterable of NetworkX graphs

rename
[iterable , optional] Node names of graphs can be changed by specifying the tuple rename=('G-','H-') (for example). Node "u" in G is then renamed "G-u" and "v" in H is renamed "H-v". Infinite generators (like itertools.count) are also supported.

Returns

U
[a graph with the same type as the first graph in list]

Raises

ValueError
If `graphs` is an empty list.

See also:

union
disjoint_union_all

Notes

To force a disjoint union with node relabeling, use disjoint_union_all(G,H) or convert_node_labels_to integers().

Graph, edge, and node attributes are propagated to the union graph. If a graph attribute is present in multiple graphs, then the value from the last graph in the list with that attribute is used.

3.45.12 disjoint_union_all

disjoint_union_all(*graphs*)

Returns the disjoint union of all graphs.

This operation forces distinct integer node labels starting with 0 for the first graph in the list and numbering consecutively.

Parameters

graphs
[iterable] Iterable of NetworkX graphs

Returns

U
[A graph with the same type as the first graph in list]

Raises

ValueError
If `graphs` is an empty list.

Notes

It is recommended that the graphs be either all directed or all undirected.

Graph, edge, and node attributes are propagated to the union graph. If a graph attribute is present in multiple graphs, then the value from the last graph in the list with that attribute is used.

3.45.13 intersection_all

intersection_all(*graphs*)

Returns a new graph that contains only the nodes and the edges that exist in all graphs.

Parameters

graphs
[iterable] Iterable of NetworkX graphs

Returns

R
[A new graph with the same type as the first graph in list]

Raises

ValueError
If `graphs` is an empty list.

Notes

Attributes from the graph, nodes, and edges are not copied to the new graph.

Graph products.

cartesian_product(G, H)	Returns the Cartesian product of G and H.
lexicographic_product(G, H)	Returns the lexicographic product of G and H.
rooted_product(G, H, root)	Return the rooted product of graphs G and H rooted at root in H.
strong_product(G, H)	Returns the strong product of G and H.
tensor_product(G, H)	Returns the tensor product of G and H.
power(G, k)	Returns the specified power of a graph.
corona_product(G, H)	Returns the Corona product of G and H.

3.45.14 cartesian_product

cartesian_product(*G*, *H*)

Returns the Cartesian product of G and H.

The Cartesian product P of the graphs G and H has a node set that is the Cartesian product of the node sets, $V(P) = V(G) \times V(H)$. P has an edge $((u, v), (x, y))$ if and only if either u is equal to x and both v and y are adjacent in H or if v is equal to y and both u and x are adjacent in G.

Parameters

G, H: graphs
Networkx graphs.

Returns

P: NetworkX graph
The Cartesian product of G and H. P will be a multi-graph if either G or H is a multi-graph. Will be a directed if G and H are directed, and undirected if G and H are undirected.

Raises

NetworkXError
If G and H are not both directed or both undirected.

Notes

Node attributes in P are two-tuple of the G and H node attributes. Missing attributes are assigned None.

Examples

```
>>> G = nx.Graph()
>>> H = nx.Graph()
>>> G.add_node(0, a1=True)
>>> H.add_node("a", a2="Spam")
>>> P = nx.cartesian_product(G, H)
>>> list(P)
[(0, 'a')]
```

Edge attributes and edge keys (for multigraphs) are also copied to the new product graph

3.45.15 lexicographic_product

lexicographic_product(*G*, *H*)

Returns the lexicographic product of G and H.

The lexicographical product P of the graphs G and H has a node set that is the Cartesian product of the node sets, $V(P) = V(G) \times V(H)$. P has an edge $((u, v), (x, y))$ if and only if (u, v) is an edge in G or $u == v$ and (x, y) is an edge in H.

Parameters

G, H: graphs
Networkx graphs.

Returns

P: NetworkX graph
The Cartesian product of G and H. P will be a multi-graph if either G or H is a multi-graph. Will be a directed if G and H are directed, and undirected if G and H are undirected.

Raises

NetworkXError
If G and H are not both directed or both undirected.

Notes

Node attributes in P are two-tuple of the G and H node attributes. Missing attributes are assigned None.

Examples

```
>>> G = nx.Graph()
>>> H = nx.Graph()
>>> G.add_node(0, a1=True)
>>> H.add_node("a", a2="Spam")
>>> P = nx.lexicographic_product(G, H)
>>> list(P)
[(0, 'a')]
```

Edge attributes and edge keys (for multigraphs) are also copied to the new product graph

3.45.16 rooted_product

rooted_product(*G*, *H*, *root*)

Return the rooted product of graphs G and H rooted at root in H.

A new graph is constructed representing the rooted product of the inputted graphs, G and H, with a root in H. A rooted product duplicates H for each nodes in G with the root of H corresponding to the node in G. Nodes are renamed as the direct product of G and H. The result is a subgraph of the cartesian product.

Parameters

G,H
[graph] A NetworkX graph

root
[node] A node in H

Returns

R
[The rooted product of G and H with a specified root in H]

Notes

The nodes of R are the Cartesian Product of the nodes of G and H. The nodes of G and H are not relabeled.

3.45.17 strong_product

strong_product(*G*, *H*)

Returns the strong product of G and H.

The strong product P of the graphs G and H has a node set that is the Cartesian product of the node sets, $V(P) = V(G) \times V(H)$. P has an edge $((u, v), (x, y))$ if and only if $u == v$ and (x, y) is an edge in H, or $x == y$ and (u, v) is an edge in G, or (u, v) is an edge in G and (x, y) is an edge in H.

Parameters

G, H: graphs
Networkx graphs.

Returns

P: NetworkX graph
The Cartesian product of G and H. P will be a multi-graph if either G or H is a multi-graph. Will be a directed if G and H are directed, and undirected if G and H are undirected.

Raises

NetworkXError
If G and H are not both directed or both undirected.

Notes

Node attributes in P are two-tuple of the G and H node attributes. Missing attributes are assigned None.

Examples

```
>>> G = nx.Graph()
>>> H = nx.Graph()
>>> G.add_node(0, a1=True)
>>> H.add_node("a", a2="Spam")
>>> P = nx.strong_product(G, H)
>>> list(P)
[(0, 'a')]
```

Edge attributes and edge keys (for multigraphs) are also copied to the new product graph

3.45.18 tensor_product

tensor_product(*G*, *H*)

Returns the tensor product of G and H.

The tensor product P of the graphs G and H has a node set that is the tensor product of the node sets, $V(P) = V(G) \times V(H)$. P has an edge $((u, v), (x, y))$ if and only if (u, x) is an edge in G and (v, y) is an edge in H.

Tensor product is sometimes also referred to as the categorical product, direct product, cardinal product or conjunction.

Parameters

G, H: graphs
Networkx graphs.

Returns

P: NetworkX graph
The tensor product of G and H. P will be a multi-graph if either G or H is a multi-graph, will be a directed if G and H are directed, and undirected if G and H are undirected.

Raises

NetworkXError
If G and H are not both directed or both undirected.

Notes

Node attributes in P are two-tuple of the G and H node attributes. Missing attributes are assigned None.

Examples

```
>>> G = nx.Graph()
>>> H = nx.Graph()
>>> G.add_node(0, a1=True)
>>> H.add_node("a", a2="Spam")
>>> P = nx.tensor_product(G, H)
>>> list(P)
[(0, 'a')]
```

Edge attributes and edge keys (for multigraphs) are also copied to the new product graph

3.45.19 power

power(G, k)

Returns the specified power of a graph.

The $k`thpowerofasimplegraph : math : `G$, denoted G^k, is a graph on the same set of nodes in which two distinct nodes u and v are adjacent in G^k if and only if the shortest path distance between u and v in G is at most k.

Parameters

G
[graph] A NetworkX simple graph object.

k
[positive integer] The power to which to raise the graph `G`.

Returns

NetworkX simple graph
`G` to the power `k`.

Raises

ValueError
If the exponent `k` is not positive.

NetworkXNotImplemented
If `G` is not a simple graph.

Notes

This definition of "power graph" comes from Exercise 3.1.6 of *Graph Theory* by Bondy and Murty [1].

References

[1]

Examples

The number of edges will never decrease when taking successive powers:

```
>>> G = nx.path_graph(4)
>>> list(nx.power(G, 2).edges)
[(0, 1), (0, 2), (1, 2), (1, 3), (2, 3)]
>>> list(nx.power(G, 3).edges)
[(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)]
```

The `k`th power of a cycle graph on *n* nodes is the complete graph on *n* nodes, if `k` is at least `n // 2`:

```
>>> G = nx.cycle_graph(5)
>>> H = nx.complete_graph(5)
>>> nx.is_isomorphic(nx.power(G, 2), H)
True
>>> G = nx.cycle_graph(8)
>>> H = nx.complete_graph(8)
>>> nx.is_isomorphic(nx.power(G, 4), H)
True
```

3.45.20 corona_product

corona_product(G, H)

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 $1 \leq i \leq |V(G)|$.

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.

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).

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
```

3.46 Planarity

check_planarity(G[, counterexample])	Check if a graph is planar and return a counterexample or an embedding.
is_planar(G)	Returns True if and only if `G` is planar.
PlanarEmbedding([incoming_graph_data])	Represents a planar graph with its planar embedding.

3.46.1 check_planarity

check_planarity(*G*, *counterexample=False*)

Check if a graph is planar and return a counterexample or an embedding.

A graph is planar iff it can be drawn in a plane without any edge intersections.

Parameters

G
[NetworkX graph]

counterexample
[bool] A Kuratowski subgraph (to proof non planarity) is only returned if set to true.

Returns

(is_planar, certificate)
[(bool, NetworkX graph) tuple] is_planar is true if the graph is planar. If the graph is planar `certificate` is a PlanarEmbedding otherwise it is a Kuratowski subgraph.

See also:

is_planar
Check for planarity without creating a *PlanarEmbedding* or counterexample.

Notes

A (combinatorial) embedding consists of cyclic orderings of the incident edges at each vertex. Given such an embedding there are multiple approaches discussed in literature to drawing the graph (subject to various constraints, e.g. integer coordinates), see e.g. [2].

The planarity check algorithm and extraction of the combinatorial embedding is based on the Left-Right Planarity Test [1].

A counterexample is only generated if the corresponding parameter is set, because the complexity of the counterexample generation is higher.

References

[1], [2]

Examples

```
>>> G = nx.Graph([(0, 1), (0, 2)])
>>> is_planar, P = nx.check_planarity(G)
>>> print(is_planar)
True
```

When G is planar, a *PlanarEmbedding* instance is returned:

```
>>> P.get_data()
{0: [1, 2], 1: [0], 2: [0]}
```

3.46.2 is_planar

is_planar(*G*)

Returns True if and only if G is planar.

A graph is *planar* iff it can be drawn in a plane without any edge intersections.

Parameters

G
[NetworkX graph]

Returns

bool
Whether the graph is planar.

See also:

check_planarity
Check if graph is planar *and* return a *PlanarEmbedding* instance if True.

Examples

```
>>> G = nx.Graph([(0, 1), (0, 2)])
>>> nx.is_planar(G)
True
>>> nx.is_planar(nx.complete_graph(5))
False
```

3.46.3 networkx.algorithms.planarity.PlanarEmbedding

class PlanarEmbedding(*incoming_graph_data=None*, ***attr*)

Represents a planar graph with its planar embedding.

The planar embedding is given by a combinatorial embedding.

Note: *check_planarity* is the preferred way to check if a graph is planar.

Neighbor ordering:

In comparison to a usual graph structure, the embedding also stores the order of all neighbors for every vertex. The order of the neighbors can be given in clockwise (cw) direction or counterclockwise (ccw) direction. This order is stored as edge attributes in the underlying directed graph. For the edge (u, v) the edge attribute 'cw' is set to the neighbor of u that follows immediately after v in clockwise direction.

In order for a PlanarEmbedding to be valid it must fulfill multiple conditions. It is possible to check if these conditions are fulfilled with the method *check_structure()*. The conditions are:

- Edges must go in both directions (because the edge attributes differ)
- Every edge must have a 'cw' and 'ccw' attribute which corresponds to a correct planar embedding.
- A node with non zero degree must have a node attribute 'first_nbr'.

As long as a PlanarEmbedding is invalid only the following methods should be called:

- *add_half_edge_ccw()*
- *add_half_edge_cw()*
- *connect_components()*
- *add_half_edge_first()*

Even though the graph is a subclass of nx.DiGraph, it can still be used for algorithms that require undirected graphs, because the method *is_directed()* is overridden. This is possible, because a valid PlanarGraph must have edges in both directions.

Half edges:

In methods like *add_half_edge_ccw* the term "half-edge" is used, which is a term that is used in doubly connected edge lists. It is used to emphasize that the edge is only in one direction and there exists another half-edge in the opposite direction. While conventional edges always have two faces (including outer face) next to them, it is possible to assign each half-edge *exactly one* face. For a half-edge (u, v) that is orientated such that u is below v then the face that belongs to (u, v) is to the right of this half-edge.

See also:

is_planar
Preferred way to check if an existing graph is planar.

check_planarity
A convenient way to create a *PlanarEmbedding*. If not planar, it returns a subgraph that shows this.

Examples

Create an embedding of a star graph (compare `nx.star_graph(3)`):

```
>>> G = nx.PlanarEmbedding()
>>> G.add_half_edge_cw(0, 1, None)
>>> G.add_half_edge_cw(0, 2, 1)
>>> G.add_half_edge_cw(0, 3, 2)
>>> G.add_half_edge_cw(1, 0, None)
>>> G.add_half_edge_cw(2, 0, None)
>>> G.add_half_edge_cw(3, 0, None)
```

Alternatively the same embedding can also be defined in counterclockwise orientation. The following results in exactly the same PlanarEmbedding:

```
>>> G = nx.PlanarEmbedding()
>>> G.add_half_edge_ccw(0, 1, None)
>>> G.add_half_edge_ccw(0, 3, 1)
>>> G.add_half_edge_ccw(0, 2, 3)
>>> G.add_half_edge_ccw(1, 0, None)
>>> G.add_half_edge_ccw(2, 0, None)
>>> G.add_half_edge_ccw(3, 0, None)
```

After creating a graph, it is possible to validate that the PlanarEmbedding object is correct:

```
>>> G.check_structure()
```

__init__(*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'}
```

Methods

add_edge(u_of_edge, v_of_edge, **attr)	Add an edge between u and v.
add_edges_from(ebunch_to_add, **attr)	Add all the edges in ebunch_to_add.
add_half_edge_ccw(start_node, end_node, ...)	Adds a half-edge from start_node to end_node.
add_half_edge_cw(start_node, end_node, ...)	Adds a half-edge from start_node to end_node.
add_half_edge_first(start_node, end_node)	The added half-edge is inserted at the first position in the order.
add_node(node_for_adding, **attr)	Add a single node `node_for_adding` and update node attributes.
add_nodes_from(nodes_for_adding, **attr)	Add multiple nodes.
add_weighted_edges_from(ebunch_to_add[, weight])	Add weighted edges in `ebunch_to_add` with specified weight attr
adjacency()	Returns an iterator over (node, adjacency dict) tuples for all nodes.
check_structure()	Runs without exceptions if this object is valid.
clear()	Remove all nodes and edges from the graph.
clear_edges()	Remove all edges from the graph without altering nodes.
connect_components(v, w)	Adds half-edges for (v, w) and (w, v) at some position.
copy([as_view])	Returns a copy of the graph.
edge_subgraph(edges)	Returns the subgraph induced by the specified edges.
get_data()	Converts the adjacency structure into a better readable structure.
get_edge_data(u, v[, default])	Returns the attribute dictionary associated with edge (u, v).
has_edge(u, v)	Returns True if the edge (u, v) is in the graph.
has_node(n)	Returns True if the graph contains the node n.
has_predecessor(u, v)	Returns True if node u has predecessor v.
has_successor(u, v)	Returns True if node u has successor v.
is_directed()	A valid PlanarEmbedding is undirected.
is_multigraph()	Returns True if graph is a multigraph, False otherwise.
nbunch_iter([nbunch])	Returns an iterator over nodes contained in nbunch that are also in the graph.
neighbors(n)	Returns an iterator over successor nodes of n.
neighbors_cw_order(v)	Generator for the neighbors of v in clockwise order.
next_face_half_edge(v, w)	Returns the following half-edge left of a face.

continues on next page

Table 1 – continued from previous page

number_of_edges([u, v])	Returns the number of edges between two nodes.
number_of_nodes()	Returns the number of nodes in the graph.
order()	Returns the number of nodes in the graph.
predecessors(n)	Returns an iterator over predecessor nodes of n.
remove_edge(u, v)	Remove the edge between u and v.
remove_edges_from(ebunch)	Remove all edges specified in ebunch.
remove_node(n)	Remove node n.
remove_nodes_from(nodes)	Remove multiple nodes.
reverse([copy])	Returns the reverse of the graph.
set_data(data)	Inserts edges according to given sorted neighbor list.
size([weight])	Returns the number of edges or total of all edge weights.
subgraph(nodes)	Returns a SubGraph view of the subgraph induced on *nodes*.
successors(n)	Returns an iterator over successor nodes of n.
to_directed([as_view])	Returns a directed representation of the graph.
to_directed_class()	Returns the class to use for empty directed copies.
to_undirected([reciprocal, as_view])	Returns an undirected representation of the digraph.
to_undirected_class()	Returns the class to use for empty undirected copies.
traverse_face(v, w[, mark_half_edges])	Returns nodes on the face that belong to the half-edge (v, w).
update([edges, nodes])	Update the graph using nodes/edges/graphs as input.

PlanarEmbedding.add_edge

PlanarEmbedding.**add_edge**(*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})
```

PlanarEmbedding.add_edges_from

`PlanarEmbedding.`**`add_edges_from`**(*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))
```

PlanarEmbedding.add_half_edge_ccw

`PlanarEmbedding.`**`add_half_edge_ccw`**(*start_node*, *end_node*, *reference_neighbor*)

Adds a half-edge from start_node to end_node.

The half-edge is added counter 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_cw
connect_components
add_half_edge_first

PlanarEmbedding.add_half_edge_cw

PlanarEmbedding.**add_half_edge_cw**(*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_ccw
connect_components
add_half_edge_first

PlanarEmbedding.add_half_edge_first

PlanarEmbedding.**add_half_edge_first**(*start_node*, *end_node*)

The added half-edge is inserted at the first position in the order.

Parameters

start_node
[node]

end_node
[node]

See also:

add_half_edge_ccw
add_half_edge_cw
connect_components

PlanarEmbedding.add_node

PlanarEmbedding.**add_node**(*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

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.

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))
```

PlanarEmbedding.add_nodes_from

PlanarEmbedding.**add_nodes_from**(*nodes_for_adding*, ***attr*)

Add multiple nodes.

Parameters

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.DiGraph([(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))
```

PlanarEmbedding.add_weighted_edges_from

PlanarEmbedding.**add_weighted_edges_from**(*ebunch_to_add*, *weight='weight'*, ***attr*)

Add weighted edges in `ebunch_to_add` with specified weight attr

Parameters

ebunch_to_add
[container of edges] Each edge given in the list or container will be added to the graph. The edges must be given as 3-tuples (u, v, w) where w is a number.

weight
[string, optional (default= 'weight')] The attribute name for the edge weights to be added.

attr
[keyword arguments, optional (default= no attributes)] Edge attributes to add/update for all edges.

See also:

add_edge
add a single edge

add_edges_from
add multiple edges

Notes

Adding the same edge twice for Graph/DiGraph simply updates the edge data. For MultiGraph/MultiDiGraph, duplicate edges are stored.

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_weighted_edges_from`.

Examples

```
>>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_weighted_edges_from([(0, 1, 3.0), (1, 2, 7.5)])
```

Evaluate an iterator over edges before passing it

```
>>> G = nx.Graph([(1, 2), (2, 3), (3, 4)])
>>> weight = 0.1
>>> # Grow graph by one new node, adding edges to all existing nodes.
>>> # wrong way - will raise RuntimeError
>>> # G.add_weighted_edges_from(((5, n, weight) for n in G.nodes))
>>> # correct way - note that there will be no self-edge for node 5
>>> G.add_weighted_edges_from(list((5, n, weight) for n in G.nodes))
```

PlanarEmbedding.adjacency

PlanarEmbedding.**adjacency**()

Returns an iterator over (node, adjacency dict) tuples for all nodes.

For directed graphs, only outgoing neighbors/adjacencies are included.

Returns

adj_iter
[iterator] An iterator over (node, adjacency dictionary) for all nodes in the graph.

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> [(n, nbrdict) for n, nbrdict in G.adjacency()]
[(0, {1: {}}), (1, {0: {}, 2: {}}), (2, {1: {}, 3: {}}), (3, {2: {}})]
```

PlanarEmbedding.check_structure

PlanarEmbedding.**check_structure**()

Runs without exceptions if this object is valid.

Checks that the following properties are fulfilled:

- Edges go in both directions (because the edge attributes differ).
- Every edge has a 'cw' and 'ccw' attribute which corresponds to a correct planar embedding.
- A node with a degree larger than 0 has a node attribute 'first_nbr'.

Running this method verifies that the underlying Graph must be planar.

Raises

NetworkXException
This exception is raised with a short explanation if the PlanarEmbedding is invalid.

PlanarEmbedding.clear

PlanarEmbedding.**clear**()

Remove all nodes and edges from the graph.

This also removes the name, and all graph, node, and edge attributes.

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.clear()
>>> list(G.nodes)
[]
>>> list(G.edges)
[]
```

PlanarEmbedding.clear_edges

PlanarEmbedding.**clear_edges**()

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)
[]
```

PlanarEmbedding.connect_components

PlanarEmbedding.**connect_components**(*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_ccw
add_half_edge_cw
add_half_edge_first

PlanarEmbedding.copy

PlanarEmbedding.**copy**(*as_view=False*)

Returns a copy of the graph.

The copy method by default returns an independent shallow copy of the graph and attributes. That is, if an attribute is a container, that container is shared by the original an the copy. Use Python's `copy.deepcopy` for new containers.

If `as_view` is True then a view is returned instead of a copy.

Parameters

as_view
[bool, optional (default=False)] If True, the returned graph-view provides a read-only view of the original graph without actually copying any data.

Returns

G
[Graph] A copy of the graph.

See also:

to_directed
return a directed copy of the graph.

Notes

All copies reproduce the graph structure, but data attributes may be handled in different ways. There are four types of copies of a graph that people might want.

Deepcopy – A "deepcopy" copies the graph structure as well as all data attributes and any objects they might contain. The entire graph object is new so that changes in the copy do not affect the original object. (see Python's copy.deepcopy)

Data Reference (Shallow) – For a shallow copy the graph structure is copied but the edge, node and graph attribute dicts are references to those in the original graph. This saves time and memory but could cause confusion if you change an attribute in one graph and it changes the attribute in the other. NetworkX does not provide this level of shallow copy.

Independent Shallow – This copy creates new independent attribute dicts and then does a shallow copy of the attributes. That is, any attributes that are containers are shared between the new graph and the original. This is exactly what `dict.copy()` provides. You can obtain this style copy using:

```
>>> G = nx.path_graph(5)
>>> H = G.copy()
>>> H = G.copy(as_view=False)
>>> H = nx.Graph(G)
>>> H = G.__class__(G)
```

Fresh Data – For fresh data, the graph structure is copied while new empty data attribute dicts are created. The resulting graph is independent of the original and it has no edge, node or graph attributes. Fresh copies are not enabled. Instead use:

```
>>> H = G.__class__()
>>> H.add_nodes_from(G)
>>> H.add_edges_from(G.edges)
```

View – Inspired by dict-views, graph-views act like read-only versions of the original graph, providing a copy of the original structure without requiring any memory for copying the information.

See the Python copy module for more information on shallow and deep copies, https://docs.python.org/3/library/copy.html.

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> H = G.copy()
```

PlanarEmbedding.edge_subgraph

PlanarEmbedding.**edge_subgraph**(*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)]
```

PlanarEmbedding.get_data

PlanarEmbedding.**get_data**()

Converts the adjacency structure into a better readable structure.

Returns

embedding
[dict] A dict mapping all nodes to a list of neighbors sorted in clockwise order.

See also:

set_data

PlanarEmbedding.get_edge_data

PlanarEmbedding.**get_edge_data**(*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
```

PlanarEmbedding.has_edge

PlanarEmbedding.**has_edge**(*u*, *v*)

Returns True if the edge (u, v) is in the graph.

This is the same as `v in G[u]` without KeyError exceptions.

Parameters

u, v
[nodes] Nodes can be, for example, strings or numbers. Nodes must be hashable (and not None) Python objects.

Returns

edge_ind
[bool] True if edge is in the graph, False otherwise.

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.has_edge(0, 1)  # using two nodes
True
>>> e = (0, 1)
>>> G.has_edge(*e)  #  e is a 2-tuple (u, v)
True
>>> e = (0, 1, {"weight": 7})
>>> G.has_edge(*e[:2])  # e is a 3-tuple (u, v, data_dictionary)
True
```

The following syntax are equivalent:

```
>>> G.has_edge(0, 1)
True
>>> 1 in G[0]  # though this gives KeyError if 0 not in G
True
```

PlanarEmbedding.has_node

`PlanarEmbedding.`**`has_node`**`(`*n*`)`

Returns True if the graph contains the node n.

Identical to `n in G`

Parameters

n
[node]

Examples

```
>>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.has_node(0)
True
```

It is more readable and simpler to use

```
>>> 0 in G
True
```

PlanarEmbedding.has_predecessor

PlanarEmbedding.**has_predecessor**(*u*, *v*)

Returns True if node u has predecessor v.

This is true if graph has the edge u<-v.

PlanarEmbedding.has_successor

PlanarEmbedding.**has_successor**(*u*, *v*)

Returns True if node u has successor v.

This is true if graph has the edge u->v.

PlanarEmbedding.is_directed

PlanarEmbedding.**is_directed**()

A valid PlanarEmbedding is undirected.

All reverse edges are contained, i.e. for every existing half-edge (v, w) the half-edge in the opposite direction (w, v) is also contained.

PlanarEmbedding.is_multigraph

PlanarEmbedding.**is_multigraph**()

Returns True if graph is a multigraph, False otherwise.

PlanarEmbedding.nbunch_iter

PlanarEmbedding.**nbunch_iter**(*nbunch=None*)

Returns an iterator over nodes contained in nbunch that are also in the graph.

The nodes in nbunch are checked for membership in the graph and if not are silently ignored.

Parameters

nbunch
[single node, container, or all nodes (default= all nodes)] The view will only report edges incident to these nodes.

Returns

niter
[iterator] An iterator over nodes in nbunch that are also in the graph. If nbunch is None, iterate over all nodes in the graph.

Raises

NetworkXError
If nbunch is not a node or sequence of nodes. If a node in nbunch is not hashable.

See also:

Graph.__iter__

Notes

When nbunch is an iterator, the returned iterator yields values directly from nbunch, becoming exhausted when nbunch is exhausted.

To test whether nbunch is a single node, one can use "if nbunch in self:", even after processing with this routine.

If nbunch is not a node or a (possibly empty) sequence/iterator or None, a `NetworkXError` is raised. Also, if any object in nbunch is not hashable, a `NetworkXError` is raised.

PlanarEmbedding.neighbors

`PlanarEmbedding.`**`neighbors`**`(`*`n`*`)`

Returns an iterator over successor nodes of n.

A successor of n is a node m such that there exists a directed edge from n to m.

Parameters

n
[node] A node in the graph

Raises

NetworkXError
If n is not in the graph.

See also:

predecessors

Notes

neighbors() and successors() are the same.

PlanarEmbedding.neighbors_cw_order

`PlanarEmbedding.`**`neighbors_cw_order`**`(`*`v`*`)`

Generator for the neighbors of v in clockwise order.

Parameters

v
[node]

Yields

node

PlanarEmbedding.next_face_half_edge

PlanarEmbedding.**next_face_half_edge**(*v*, *w*)

Returns the following half-edge left of a face.

Parameters

v
[node]

w
[node]

Returns

half-edge
[tuple]

PlanarEmbedding.number_of_edges

PlanarEmbedding.**number_of_edges**(*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
```

PlanarEmbedding.number_of_nodes

PlanarEmbedding.**number_of_nodes**()

Returns the number of nodes in the graph.

Returns

nnodes
[int] The number of nodes in the graph.

See also:

order
identical method

__len__
identical method

Examples

```
>>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.number_of_nodes()
3
```

PlanarEmbedding.order

PlanarEmbedding.**order**()

Returns the number of nodes in the graph.

Returns

nnodes
[int] The number of nodes in the graph.

See also:

number_of_nodes
identical method

__len__
identical method

Examples

```
>>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.order()
3
```

PlanarEmbedding.predecessors

PlanarEmbedding.**predecessors**(*n*)

Returns an iterator over predecessor nodes of n.

A predecessor of n is a node m such that there exists a directed edge from m to n.

Parameters

n
[node] A node in the graph

Raises

NetworkXError
If n is not in the graph.

See also:

successors

PlanarEmbedding.remove_edge

PlanarEmbedding.**remove_edge**(*u*, *v*)

Remove the edge between u and v.

Parameters

u, v
[nodes] Remove the edge between nodes u and v.

Raises

NetworkXError
If there is not an edge between u and v.

See also:

remove_edges_from
remove a collection of edges

Examples

```
>>> G = nx.Graph()  # or DiGraph, etc
>>> nx.add_path(G, [0, 1, 2, 3])
>>> G.remove_edge(0, 1)
>>> e = (1, 2)
>>> G.remove_edge(*e)  # unpacks e from an edge tuple
>>> e = (2, 3, {"weight": 7})  # an edge with attribute data
>>> G.remove_edge(*e[:2])  # select first part of edge tuple
```

PlanarEmbedding.remove_edges_from

PlanarEmbedding.**remove_edges_from**(*ebunch*)

Remove all edges specified in ebunch.

Parameters

ebunch: list or container of edge tuples
Each edge given in the list or container will be removed from the graph. The edges can be:

- 2-tuples (u, v) edge between u and v.
- 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)
```

PlanarEmbedding.remove_node

PlanarEmbedding.**remove_node**(*n*)

Remove node n.

Removes the node n and all adjacent edges. Attempting to remove a non-existent node will raise an exception.

Parameters

n
[node] A node in the graph

Raises

NetworkXError
If n is not in the graph.

See also:

remove_nodes_from

Examples

```
>>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> list(G.edges)
[(0, 1), (1, 2)]
>>> G.remove_node(1)
>>> list(G.edges)
[]
```

PlanarEmbedding.remove_nodes_from

PlanarEmbedding.**remove_nodes_from**(*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`.

Examples

```
>>> G = nx.path_graph(3)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> e = list(G.nodes)
>>> e
[0, 1, 2]
>>> G.remove_nodes_from(e)
>>> list(G.nodes)
[]
```

Evaluate an iterator over a graph if using it to modify the same graph

```
>>> G = nx.DiGraph([(0, 1), (1, 2), (3, 4)])
>>> # this command will fail, as the graph's dict is modified during iteration
>>> # G.remove_nodes_from(n for n in G.nodes if n < 2)
>>> # this command will work, since the dictionary underlying graph is not␣
```

(continues on next page)

(continued from previous page)

```
↪modified
>>> G.remove_nodes_from(list(n for n in G.nodes if n < 2))
```

PlanarEmbedding.reverse

PlanarEmbedding.**reverse**(*copy=True*)

Returns the reverse of the graph.

The reverse is a graph with the same nodes and edges but with the directions of the edges reversed.

Parameters

copy
[bool optional (default=True)] If True, return a new DiGraph holding the reversed edges. If False, the reverse graph is created using a view of the original graph.

PlanarEmbedding.set_data

PlanarEmbedding.**set_data**(*data*)

Inserts edges according to given sorted neighbor list.

The input format is the same as the output format of get_data().

Parameters

data
[dict] A dict mapping all nodes to a list of neighbors sorted in clockwise order.

See also:

get_data

PlanarEmbedding.size

PlanarEmbedding.**size**(*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
```

PlanarEmbedding.subgraph

PlanarEmbedding.**subgraph**(*nodes*)

Returns a SubGraph view of the subgraph induced on *nodes*.

The induced subgraph of the graph contains the nodes in *nodes* and the edges between those nodes.

Parameters

nodes
[list, iterable] A container of nodes which will be iterated through once.

Returns

G
[SubGraph View] A subgraph view of the graph. The graph structure cannot be changed but node/edge attributes can and are shared with the original graph.

Notes

The graph, edge and node attributes are shared with the original graph. Changes to the graph structure is ruled out by the view, but changes to attributes are reflected in the original graph.

To create a subgraph with its own copy of the edge/node attributes use: G.subgraph(nodes).copy()

For an inplace reduction of a graph to a subgraph you can remove nodes: G.remove_nodes_from([n for n in G if n not in set(nodes)])

Subgraph views are sometimes NOT what you want. In most cases where you want to do more than simply look at the induced edges, it makes more sense to just create the subgraph as its own graph with code like:

```
# Create a subgraph SG based on a (possibly multigraph) G
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
```

(continues on next page)

(continued from previous page)

```
        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)]
```

PlanarEmbedding.successors

PlanarEmbedding.**successors**(*n*)

Returns an iterator over successor nodes of n.

A successor of n is a node m such that there exists a directed edge from n to m.

Parameters

n
[node] A node in the graph

Raises

NetworkXError
If n is not in the graph.

See also:

predecessors

Notes

neighbors() and successors() are the same.

PlanarEmbedding.to_directed

PlanarEmbedding.**to_directed**(*as_view=False*)

Returns a directed representation of the graph.

Returns

G
[DiGraph] A directed graph with the same name, same nodes, and with each edge (u, v, data) replaced by two directed edges (u, v, data) and (v, u, data).

Notes

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 D=DiGraph(G) 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 Graph to use dict-like objects in the data structure, those changes do not transfer to the DiGraph created by this method.

Examples

```
>>> G = nx.Graph()  # or MultiGraph, etc
>>> G.add_edge(0, 1)
>>> H = G.to_directed()
>>> list(H.edges)
[(0, 1), (1, 0)]
```

If already directed, return a (deep) copy

```
>>> G = nx.DiGraph()  # or MultiDiGraph, etc
>>> G.add_edge(0, 1)
>>> H = G.to_directed()
>>> list(H.edges)
[(0, 1)]
```

PlanarEmbedding.to_directed_class

PlanarEmbedding.**to_directed_class**()

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.

PlanarEmbedding.to_undirected

PlanarEmbedding.**to_undirected**(*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)]
```

PlanarEmbedding.to_undirected_class

PlanarEmbedding.**to_undirected_class**()

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.

PlanarEmbedding.traverse_face

PlanarEmbedding.**traverse_face**(*v*, *w*, *mark_half_edges=None*)

Returns nodes on the face that belong to the half-edge (v, w).

The face that is traversed lies to the right of the half-edge (in an orientation where v is below w).

Optionally it is possible to pass a set to which all encountered half edges are added. Before calling this method, this set must not include any half-edges that belong to the face.

Parameters

v
[node] Start node of half-edge.

w
[node] End node of half-edge.

mark_half_edges: set, optional
Set to which all encountered half-edges are added.

Returns

face
[list] A list of nodes that lie on this face.

PlanarEmbedding.update

`PlanarEmbedding.`**`update`**(*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.

See also:

add_edges_from
add multiple edges to a graph

add_nodes_from
add multiple nodes to a graph

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}}
>>> e = [
...     (u, v, {"weight": d})
...     for u, nbrs in adj.items()
...     for v, d in nbrs.items()
... ]
>>> DG.update(edges=e, nodes=adj)
```

```
>>> # dict-of-dict-of-dict
>>> adj = {1: {2: {"weight": 1.3}, 3: {"color": 0.7, "weight": 1.2}}}
>>> e = [
...     (u, v, {"weight": d})
...     for u, nbrs in adj.items()
...     for v, d in nbrs.items()
... ]
>>> DG.update(edges=e, nodes=adj)
```

```
>>> # predecessor adjacency (dict-of-set)
>>> pred = {1: {2, 3}, 2: {3}, 3: {3}}
>>> e = [(v, u) for u, nbrs in pred.items() for v in nbrs]
```

```
>>> # MultiGraph dict-of-dict-of-dict-of-attribute
>>> MDG = nx.MultiDiGraph()
>>> adj = {
...     1: {2: {0: {"weight": 1.3}, 1: {"weight": 1.2}}},
...     3: {2: {0: {"weight": 0.7}}},
... }
>>> e = [
...     (u, v, ekey, d)
...     for u, nbrs in adj.items()
...     for v, keydict in nbrs.items()
...     for ekey, d in keydict.items()
... ]
>>> MDG.update(edges=e)
```

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)
```

Attributes

adj	Graph adjacency object holding the neighbors of each node.
degree	A DegreeView for the Graph as G.degree or G.degree().
edges	An OutEdgeView of the DiGraph as G.edges or G.edges().
in_degree	An InDegreeView for (node, in_degree) or in_degree for single node.
in_edges	A view of the in edges of the graph as G.in_edges or G.in_edges().
name	String identifier of the graph.
nodes	A NodeView of the Graph as G.nodes or G.nodes().
out_degree	An OutDegreeView for (node, out_degree)
out_edges	An OutEdgeView of the DiGraph as G.edges or G.edges().
pred	Graph adjacency object holding the predecessors of each node.
succ	Graph adjacency object holding the successors of each node.

PlanarEmbedding.adj

property PlanarEmbedding.**adj**

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.

PlanarEmbedding.degree

property PlanarEmbedding.**degree**

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

DiDegreeView or int
If multiple nodes are requested (the default), returns a `DiDegreeView` mapping nodes to their degree. If a single node is requested, returns the degree of the node as an integer.

See also:

in_degree*, *out_degree

Examples

```
>>> G = nx.DiGraph()  # or 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)]
```

PlanarEmbedding.edges

property PlanarEmbedding.**edges**

An OutEdgeView of the DiGraph as G.edges or G.edges().

edges(self, nbunch=None, data=False, default=None)

The OutEdgeView provides set-like operations on the edge-tuples as well as edge attribute lookup. When called, it also provides an EdgeDataView object which allows control of access to edge attributes (but does not provide set-like operations). Hence, `G.edges[u, v]['color']` provides the value of the color attribute for edge `(u, v)` while `for (u, v, c) in G.edges.data('color', default='red'):` iterates through all the edges yielding the color attribute with default `'red'` if no color attribute exists.

Parameters

nbunch
[single node, container, or all nodes (default= all nodes)] The view will only report edges from 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).

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

edges
[OutEdgeView] 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']`.

See also:

in_edges, *out_edges*

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.DiGraph()  # or MultiDiGraph, etc
>>> nx.add_path(G, [0, 1, 2])
>>> 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)
OutEdgeDataView([(0, 1, {}), (1, 2, {}), (2, 3, {'weight': 5})])
>>> G.edges.data("weight", default=1)
OutEdgeDataView([(0, 1, 1), (1, 2, 1), (2, 3, 5)])
>>> G.edges([0, 2])  # only edges originating from these nodes
OutEdgeDataView([(0, 1), (2, 3)])
>>> G.edges(0)  # only edges from node 0
OutEdgeDataView([(0, 1)])
```

PlanarEmbedding.in_degree

property PlanarEmbedding.**in_degree**

An InDegreeView for (node, in_degree) or in_degree for single node.

The node in_degree is the number of edges pointing to the node. The weighted node degree is the sum of the edge weights for edges incident to that node.

This object provides an iteration over (node, in_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

If a single node is requested
deg
[int] In-degree of the node

OR if multiple nodes are requested
nd_iter
[iterator] The iterator returns two-tuples of (node, in-degree).

See also:

degree, *out_degree*

Examples

```
>>> G = nx.DiGraph()
>>> nx.add_path(G, [0, 1, 2, 3])
>>> G.in_degree(0)  # node 0 with degree 0
0
>>> list(G.in_degree([0, 1, 2]))
[(0, 0), (1, 1), (2, 1)]
```

PlanarEmbedding.in_edges

property PlanarEmbedding.**in_edges**

A view of the in edges of the graph as G.in_edges or G.in_edges().

in_edges(self, nbunch=None, data=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).

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
[InEdgeView or InEdgeDataView] 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']`.

See also:

edges

Examples

```
>>> G = nx.DiGraph()
>>> G.add_edge(1, 2, color='blue')
>>> G.in_edges()
InEdgeView([(1, 2)])
>>> G.in_edges(nbunch=2)
InEdgeDataView([(1, 2)])
```

PlanarEmbedding.name

property PlanarEmbedding.**name**

String identifier of the graph.

This graph attribute appears in the attribute dict G.graph keyed by the string `"name"`. as well as an attribute (technically a property) `G.name`. This is entirely user controlled.

PlanarEmbedding.nodes

property PlanarEmbedding.**nodes**

A NodeView of the Graph as G.nodes or G.nodes().

Can be used as `G.nodes` for data lookup and for set-like operations. Can also be used as `G.nodes(data='color', default=None)` to return a NodeDataView which reports specific node data but no set operations. It presents a dict-like interface as well with `G.nodes.items()` iterating over `(node, nodedata)` 2-tuples and `G.nodes[3]['foo']` providing the value of the `foo` attribute for node `3`. In addition, a view `G.nodes.data('foo')` provides a dict-like interface to the `foo` attribute of each node. `G.nodes.data('foo', default=1)` provides a default for nodes that do not have attribute `foo`.

Parameters

data
[string or bool, optional (default=False)] The node attribute returned in 2-tuple (n, ddict[data]). If True, return entire node attribute dict as (n, ddict). If False, return just the nodes n.

default
[value, optional (default=None)] Value used for nodes that don't have the requested attribute. Only relevant if data is not True or False.

Returns

NodeView
Allows set-like operations over the nodes as well as node attribute dict lookup and calling to get a NodeDataView. A NodeDataView iterates over `(n, data)` and has no set operations. A NodeView iterates over `n` and includes set operations.

When called, if data is False, an iterator over nodes. Otherwise an iterator of 2-tuples (node, attribute value) where the attribute is specified in `data`. If data is True then the attribute becomes the entire data dictionary.

Notes

If your node data is not needed, it is simpler and equivalent to use the expression `for n in G`, or `list(G)`.

Examples

There are two simple ways of getting a list of all nodes in the graph:

```
>>> G = nx.path_graph(3)
>>> list(G.nodes)
[0, 1, 2]
>>> list(G)
[0, 1, 2]
```

To get the node data along with the nodes:

```
>>> G.add_node(1, time="5pm")
>>> G.nodes[0]["foo"] = "bar"
>>> list(G.nodes(data=True))
[(0, {'foo': 'bar'}), (1, {'time': '5pm'}), (2, {})]
>>> list(G.nodes.data())
[(0, {'foo': 'bar'}), (1, {'time': '5pm'}), (2, {})]
```

```
>>> list(G.nodes(data="foo"))
[(0, 'bar'), (1, None), (2, None)]
>>> list(G.nodes.data("foo"))
[(0, 'bar'), (1, None), (2, None)]
```

```
>>> list(G.nodes(data="time"))
[(0, None), (1, '5pm'), (2, None)]
>>> list(G.nodes.data("time"))
[(0, None), (1, '5pm'), (2, None)]
```

```
>>> list(G.nodes(data="time", default="Not Available"))
[(0, 'Not Available'), (1, '5pm'), (2, 'Not Available')]
>>> list(G.nodes.data("time", default="Not Available"))
[(0, 'Not Available'), (1, '5pm'), (2, 'Not Available')]
```

If some of your nodes have an attribute and the rest are assumed to have a default attribute value you can create a dictionary from node/attribute pairs using the `default` keyword argument to guarantee the value is never None:

```
>>> G = nx.Graph()
>>> G.add_node(0)
>>> G.add_node(1, weight=2)
>>> G.add_node(2, weight=3)
>>> dict(G.nodes(data="weight", default=1))
{0: 1, 1: 2, 2: 3}
```

PlanarEmbedding.out_degree

property PlanarEmbedding.**out_degree**

An OutDegreeView for (node, out_degree)

The node out_degree is the number of edges pointing out of the node. The weighted node degree is the sum of the edge weights for edges incident to that node.

This object provides an iterator over (node, out_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

If a single node is requested
deg
[int] Out-degree of the node

OR if multiple nodes are requested
nd_iter
[iterator] The iterator returns two-tuples of (node, out-degree).

See also:

degree, *in_degree*

Examples

```
>>> G = nx.DiGraph()
>>> nx.add_path(G, [0, 1, 2, 3])
>>> G.out_degree(0)  # node 0 with degree 1
1
>>> list(G.out_degree([0, 1, 2]))
[(0, 1), (1, 1), (2, 1)]
```

PlanarEmbedding.out_edges

property PlanarEmbedding.**out_edges**

An OutEdgeView of the DiGraph as G.edges or G.edges().

edges(self, nbunch=None, data=False, default=None)

The OutEdgeView provides set-like operations on the edge-tuples as well as edge attribute lookup. When called, it also provides an EdgeDataView object which allows control of access to edge attributes (but does not provide set-like operations). Hence, `G.edges[u, v]['color']` provides the value of the color attribute for edge `(u, v)` while `for (u, v, c) in G.edges.data('color', default='red'):` iterates through all the edges yielding the color attribute with default `'red'` if no color attribute exists.

Parameters

nbunch
[single node, container, or all nodes (default= all nodes)] The view will only report edges from 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).

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

edges
[OutEdgeView] 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']`.

See also:

in_edges*, *out_edges

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.DiGraph()  # or MultiDiGraph, etc
>>> nx.add_path(G, [0, 1, 2])
>>> 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)
OutEdgeDataView([(0, 1, {}), (1, 2, {}), (2, 3, {'weight': 5})])
>>> G.edges.data("weight", default=1)
OutEdgeDataView([(0, 1, 1), (1, 2, 1), (2, 3, 5)])
>>> G.edges([0, 2])  # only edges originating from these nodes
OutEdgeDataView([(0, 1), (2, 3)])
>>> G.edges(0)  # only edges from node 0
OutEdgeDataView([(0, 1)])
```

PlanarEmbedding.pred

property PlanarEmbedding.**pred**

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.

PlanarEmbedding.succ

property PlanarEmbedding.**succ**

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`.

3.47 Planar Drawing

combinatorial_embedding_to_pos(embedding[, ...])	Assigns every node a (x, y) position based on the given embedding

3.47.1 combinatorial_embedding_to_pos

combinatorial_embedding_to_pos(*embedding*, *fully_triangulate=False*)

Assigns every node a (x, y) position based on the given embedding

The algorithm iteratively inserts nodes of the input graph in a certain order and rearranges previously inserted nodes so that the planar drawing stays valid. This is done efficiently by only maintaining relative positions during the node placements and calculating the absolute positions at the end. For more information see [1].

Parameters

embedding
[nx.PlanarEmbedding] This defines the order of the edges

fully_triangulate
[bool] If set to True the algorithm adds edges to a copy of the input embedding and makes it chordal.

Returns

pos
[dict] Maps each node to a tuple that defines the (x, y) position

References

[1]

3.48 Graph Polynomials

Provides algorithms supporting the computation of graph polynomials.

Graph polynomials are polynomial-valued graph invariants that encode a wide variety of structural information. Examples include the Tutte polynomial, chromatic polynomial, characteristic polynomial, and matching polynomial. An extensive treatment is provided in [1].

For a simple example, the `charpoly` 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
```

tutte_polynomial(G)	Returns the Tutte polynomial of `G`
chromatic_polynomial(G)	Returns the chromatic polynomial of `G`

3.48.1 tutte_polynomial

tutte_polynomial(*G*)

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]:

$$T_G(x, y) = \sum_{A \in E} (x-1)^{c(A)-c(E)} (y-1)^{c(A)+|A|-n(G)}$$

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]:

$$T_G(x, y) = \sum_{T \text{ a spanning tree of } G} x^{i(T)} y^{e(T)}$$

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`:

$$T_G(x, y) = \begin{cases} x^{k(G)} y^{l(G)}, & \text{if all edges are cut-edges or self-loops} \\ T_{G-e}(x, y) + T_{G/e}(x, y), & \text{otherwise, for an arbitrary edge } e \text{ not a cut-edge or loop} \end{cases}$$

Parameters

G
[NetworkX graph]

Returns

instance of `sympy.core.add.Add`
A Sympy expression representing the Tutte polynomial for `G`.

Notes

Some specializations of the Tutte polynomial:

- `T_G(1, 1)` counts the number of spanning trees of `G`
- `T_G(1, 2)` counts the number of connected spanning subgraphs of `G`
- `T_G(2, 1)` counts the number of spanning forests in `G`
- `T_G(0, 2)` counts the number of strong orientations of `G`
- `T_G(2, 0)` counts the number of acyclic orientations of `G`

Edge contraction is defined and deletion-contraction is introduced in [6]. Combinatorial meaning of the coefficients is introduced in [7]. Universality, properties, and applications are discussed in [8].

Practically, up-front computation of the Tutte polynomial may be useful when users wish to repeatedly calculate edge-connectivity-related information about one or more graphs.

References

[1], [2], [3], [4], [5], [6], [7], [8]

Examples

```
>>> C = nx.cycle_graph(5)
>>> nx.tutte_polynomial(C)
x**4 + x**3 + x**2 + x + y
```

```
>>> D = nx.diamond_graph()
>>> nx.tutte_polynomial(D)
x**3 + 2*x**2 + 2*x*y + x + y**2 + y
```

3.48.2 chromatic_polynomial

chromatic_polynomial(*G*)

Returns the chromatic polynomial of `G`

This function computes the chromatic polynomial via an iterative version of the deletion-contraction algorithm.

The chromatic polynomial `X_G(x)` is a fundamental graph polynomial invariant in one variable. Evaluating `X_G(k)` for an natural number `k` enumerates the proper k-colorings of `G`.

There are several equivalent definitions; here are three:

Def 1 (explicit formula): For `G` an undirected graph, `c(G)` the number of connected components of `G`, `E` the edge set of `G`, and `G(S)` the spanning subgraph of `G` with edge set `S` [1]:

$$X_G(x) = \sum_{S \subseteq E} (-1)^{|S|} x^{c(G(S))}$$

Def 2 (interpolating polynomial): For `G` an undirected graph, `n(G)` the number of vertices of `G`, `k_0 = 0`, and `k_i` the number of distinct ways to color the vertices of `G` with `i` unique colors (for `i` a natural number at most `n(G)`), `X_G(x)` is the unique Lagrange interpolating polynomial of degree `n(G)` through the points `(0, k_0), (1, k_1), dots, (n(G), k_{n(G)})` [2].

Def 3 (chromatic 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`, `n(G)` the number of vertices of `G`, and `e(G)` the number of edges of `G` [3]:

$$X_G(x) = \begin{cases} x^{n(G)}, & \text{if } e(G) = 0 \\ X_{G-e}(x) - X_{G/e}(x), & \text{otherwise, for an arbitrary edge } e \end{cases}$$

This formulation is also known as the Fundamental Reduction Theorem [4].

Parameters

G
[NetworkX graph]

Returns

instance of `sympy.core.add.Add`
A Sympy expression representing the chromatic polynomial for `G`.

Notes

Interpretation of the coefficients is discussed in [5]. Several special cases are listed in [2].

The chromatic polynomial is a specialization of the Tutte polynomial; in particular, `X_G(x) = `T_G(x, 0)` [6].

The chromatic polynomial may take negative arguments, though evaluations may not have chromatic interpretations. For instance, `X_G(-1)` enumerates the acyclic orientations of `G` [7].

References

[1], [2], [3], [4], [5], [6], [7]

Examples

```
>>> C = nx.cycle_graph(5)
>>> nx.chromatic_polynomial(C)
x**5 - 5*x**4 + 10*x**3 - 10*x**2 + 4*x
```

```
>>> G = nx.complete_graph(4)
>>> nx.chromatic_polynomial(G)
x**4 - 6*x**3 + 11*x**2 - 6*x
```

3.49 Reciprocity

Algorithms to calculate reciprocity in a directed graph.

reciprocity(G[, nodes])	Compute the reciprocity in a directed graph.
overall_reciprocity(G)	Compute the reciprocity for the whole graph.

3.49.1 reciprocity

reciprocity(*G*, *nodes=None*)

Compute the reciprocity in a directed graph.

The reciprocity of a directed graph is defined as the ratio of the number of edges pointing in both directions to the total number of edges in the graph. Formally, $r = |(u, v) \in G | (v, u) \in G| / |(u, v) \in G|$.

The reciprocity of a single node u is defined similarly, it is the ratio of the number of edges in both directions to the total number of edges attached to node u.

Parameters

G
[graph] A networkx directed graph

nodes
[container of nodes, optional (default=whole graph)] Compute reciprocity for nodes in this container.

Returns

out
[dictionary] Reciprocity keyed by node label.

Notes

The reciprocity is not defined for isolated nodes. In such cases this function will return None.

3.49.2 overall_reciprocity

overall_reciprocity(*G*)

Compute the reciprocity for the whole graph.

See the doc of reciprocity for the definition.

Parameters

G
[graph] A networkx graph

3.50 Regular

Functions for computing and verifying regular graphs.

is_regular(G)	Determines whether the graph G is a regular graph.
is_k_regular(G, k)	Determines whether the graph G is a k-regular graph.
k_factor(G, k[, matching_weight])	Compute a k-factor of G

3.50.1 is_regular

is_regular(*G*)

Determines whether the graph G is a regular graph.

A regular graph is a graph where each vertex has the same degree. A regular digraph is a graph where the indegree and outdegree of each vertex are equal.

Parameters

G
[NetworkX graph]

Returns

bool
Whether the given graph or digraph is regular.

Examples

```
>>> G = nx.DiGraph([(1, 2), (2, 3), (3, 4), (4, 1)])
>>> nx.is_regular(G)
True
```

3.50.2 is_k_regular

is_k_regular(*G*, *k*)

Determines whether the graph G is a k-regular graph.

A k-regular graph is a graph where each vertex has degree k.

Parameters

G
[NetworkX graph]

Returns

bool
Whether the given graph is k-regular.

Examples

```
>>> G = nx.Graph([(1, 2), (2, 3), (3, 4), (4, 1)])
>>> nx.is_k_regular(G, k=3)
False
```

3.50.3 k_factor

k_factor(*G*, *k*, *matching_weight='weight'*)

Compute a k-factor of G

A k-factor of a graph is a spanning k-regular subgraph. A spanning k-regular subgraph of G is a subgraph that contains each vertex of G and a subset of the edges of G such that each vertex has degree k.

Parameters

G
[NetworkX graph] Undirected graph

matching_weight: string, optional (default='weight')
Edge data key corresponding to the edge weight. Used for finding the max-weighted perfect matching. If key not found, uses 1 as weight.

Returns

G2
[NetworkX graph] A k-factor of G

References

[1]

Examples

```
>>> G = nx.Graph([(1, 2), (2, 3), (3, 4), (4, 1)])
>>> G2 = nx.k_factor(G, k=1)
>>> G2.edges()
EdgeView([(1, 2), (3, 4)])
```

3.51 Rich Club

Functions for computing rich-club coefficients.

rich_club_coefficient(G[, normalized, Q, seed])	Returns the rich-club coefficient of the graph G.

3.51.1 rich_club_coefficient

rich_club_coefficient(*G*, *normalized=True*, *Q=100*, *seed=None*)

Returns the rich-club coefficient of the graph G.

For each degree *k*, the *rich-club coefficient* is the ratio of the number of actual to the number of potential edges for nodes with degree greater than *k*:

$$\phi(k) = \frac{2E_k}{N_k(N_k - 1)}$$

where N_k is the number of nodes with degree larger than *k*, and E_k is the number of edges among those nodes.

Parameters

G
[NetworkX graph] Undirected graph with neither parallel edges nor self-loops.

normalized
[bool (optional)] Normalize using randomized network as in [1]

Q
[float (optional, default=100)] If normalized is True, perform Q * m double-edge swaps, where m is the number of edges in G, to use as a null-model for normalization.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

rc
[dictionary] A dictionary, keyed by degree, with rich-club coefficient values.

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.

Estimates for appropriate values of Q are found in [2].

References

[1], [2]

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
```

3.52 Shortest Paths

Compute the shortest paths and path lengths between nodes in the graph.

These algorithms work with undirected and directed graphs.

shortest_path(G[, source, target, weight, ...])	Compute shortest paths in the graph.
all_shortest_paths(G, source, target[, ...])	Compute all shortest simple paths in the graph.
shortest_path_length(G[, source, target, ...])	Compute shortest path lengths in the graph.
average_shortest_path_length(G[, weight, method])	Returns the average shortest path length.
has_path(G, source, target)	Returns *True* if *G* has a path from *source* to *target*.

3.52.1 shortest_path

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.

See also:

all_pairs_shortest_path
all_pairs_dijkstra_path
all_pairs_bellman_ford_path
single_source_shortest_path
single_source_dijkstra_path
single_source_bellman_ford_path

Notes

There may be more than one shortest path between a source and target. This returns only one of them.

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 = nx.shortest_path(G)  # source, target not specified
>>> p[2][4] # shortest path from source=2 to target=4
[2, 3, 4]
```

3.52.2 all_shortest_paths

all_shortest_paths(*G*, *source*, *target*, *weight=None*, *method='dijkstra'*)

Compute all shortest simple paths in the graph.

Parameters

G
[NetworkX graph]

source
[node] Starting node for path.

target
[node] Ending 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 lists] A generator of all paths between source and target.

Raises

ValueError
If `method` is not among the supported options.

NetworkXNoPath
If `target` cannot be reached from `source`.

See also:

shortest_path
single_source_shortest_path
all_pairs_shortest_path

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.

Examples

```
>>> G = nx.Graph()
>>> nx.add_path(G, [0, 1, 2])
>>> nx.add_path(G, [0, 10, 2])
>>> print([p for p in nx.all_shortest_paths(G, source=0, target=2)])
[[0, 1, 2], [0, 10, 2]]
```

3.52.3 shortest_path_length

shortest_path_length(*G*, *source=None*, *target=None*, *weight=None*, *method='dijkstra'*)

Compute shortest path lengths in the graph.

Parameters

G
[NetworkX graph]

source
[node, optional] Starting node for path. If not specified, compute shortest path lengths using all nodes as source nodes.

target
[node, optional] Ending node for path. If not specified, compute shortest path lengths using all nodes as target 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 length. 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

length: int or iterator
If the source and target are both specified, return the length of the shortest path from the source to the target.

If only the source is specified, return a dict keyed by target to the shortest path length from the source to that target.

If only the target is specified, return a dict keyed by source to the shortest path length from that source to the target.

If neither the source nor target are specified, return an iterator over (source, dictionary) where dictionary is keyed by target to shortest path length from source to that target.

Raises

NodeNotFound
If `source` is not in `G`.

NetworkXNoPath
If no path exists between source and target.

ValueError
If `method` is not among the supported options.

See also:

all_pairs_shortest_path_length
all_pairs_dijkstra_path_length
all_pairs_bellman_ford_path_length
single_source_shortest_path_length
single_source_dijkstra_path_length
single_source_bellman_ford_path_length

Notes

The length of the path is always 1 less than the number of nodes involved in the path since the length measures the number of edges followed.

For digraphs this returns the shortest directed path length. To find path lengths in the reverse direction use G.reverse(copy=False) first to flip the edge orientation.

Examples

```
>>> G = nx.path_graph(5)
>>> nx.shortest_path_length(G, source=0, target=4)
4
>>> p = nx.shortest_path_length(G, source=0)  # target not specified
>>> p[4]
4
>>> p = nx.shortest_path_length(G, target=4)  # source not specified
>>> p[0]
4
>>> p = dict(nx.shortest_path_length(G))  # source,target not specified
>>> p[0][4]
4
```

3.52.4 average_shortest_path_length

average_shortest_path_length(*G*, *weight=None*, *method=None*)

Returns the average shortest path length.

The average shortest path length is

$$a = \sum_{\substack{s,t \in V \\ s \neq t}} \frac{d(s,t)}{n(n-1)}$$

where `V` is the set of nodes in `G`, `d(s, t)` is the shortest path from `s` to `t`, and `n` is the number of nodes in `G`.

Changed in version 3.0: An exception is raised for directed graphs that are not strongly connected.

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. 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 = 'unweighted' or 'djikstra')] The algorithm to use to compute the path lengths. Supported options are 'unweighted', 'dijkstra', 'bellman-ford', 'floyd-warshall' and 'floyd-warshall-numpy'. Other method values produce a ValueError. The default method is 'unweighted' if `weight` is None, otherwise the default method is 'dijkstra'.

Raises

NetworkXPointlessConcept
If `G` is the null graph (that is, the graph on zero nodes).

NetworkXError
If `G` is not connected (or not strongly connected, in the case of a directed graph).

ValueError
If `method` is not among the supported options.

Examples

```
>>> G = nx.path_graph(5)
>>> nx.average_shortest_path_length(G)
2.0
```

For disconnected graphs, you can compute the average shortest path length for each component

```
>>> G = nx.Graph([(1, 2), (3, 4)])
>>> for C in (G.subgraph(c).copy() for c in nx.connected_components(G)):
...     print(nx.average_shortest_path_length(C))
1.0
1.0
```

3.52.5 has_path

has_path(*G*, *source*, *target*)

Returns *True* if *G* has a path from *source* to *target*.

Parameters

G
[NetworkX graph]

source
[node] Starting node for path

target
[node] Ending node for path

3.52.6 Advanced Interface

Shortest path algorithms for unweighted graphs.

single_source_shortest_path(G, source[, cutoff])	Compute shortest path between source and all other nodes reachable from source.
single_source_shortest_path_length(G, source)	Compute the shortest path lengths from source to all reachable nodes.
single_target_shortest_path(G, target[, cutoff])	Compute shortest path to target from all nodes that reach target.
single_target_shortest_path_length(G, target)	Compute the shortest path lengths to target from all reachable nodes.
bidirectional_shortest_path(G, source, target)	Returns a list of nodes in a shortest path between source and target.
all_pairs_shortest_path(G[, cutoff])	Compute shortest paths between all nodes.
all_pairs_shortest_path_length(G[, cutoff])	Computes the shortest path lengths between all nodes in G.
predecessor(G, source[, target, cutoff, ...])	Returns dict of predecessors for the path from source to all nodes in G.

single_source_shortest_path

single_source_shortest_path(*G*, *source*, *cutoff=None*)

Compute shortest path between source and all other nodes reachable from source.

Parameters

G
[NetworkX graph]

source
[node label] Starting node for path

cutoff
[integer, optional] Depth to stop the search. Only paths of length <= cutoff are returned.

Returns

lengths
[dictionary] Dictionary, keyed by target, of shortest paths.

See also:

shortest_path

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.

Examples

```
>>> G = nx.path_graph(5)
>>> path = nx.single_source_shortest_path(G, 0)
>>> path[4]
[0, 1, 2, 3, 4]
```

single_source_shortest_path_length

single_source_shortest_path_length(*G*, *source*, *cutoff=None*)

Compute the shortest path lengths from source to all reachable nodes.

Parameters

G
[NetworkX graph]

source
[node] Starting node for path

cutoff
[integer, optional] Depth to stop the search. Only paths of length <= cutoff are returned.

Returns

lengths
[dict] Dict keyed by node to shortest path length to source.

See also:

shortest_path_length

Examples

```
>>> G = nx.path_graph(5)
>>> length = nx.single_source_shortest_path_length(G, 0)
>>> length[4]
4
>>> for node in length:
...     print(f"{node}: {length[node]}")
0: 0
1: 1
2: 2
3: 3
4: 4
```

single_target_shortest_path

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

lengths
[dictionary] Dictionary, keyed by target, of shortest paths.

See also:

shortest_path, ***single_source_shortest_path***

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.

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]
```

single_target_shortest_path_length

single_target_shortest_path_length(*G*, *target*, *cutoff=None*)

Compute the shortest path lengths to target from all reachable nodes.

Parameters

G
[NetworkX graph]

target
[node] Target node for path

cutoff
[integer, optional] Depth to stop the search. Only paths of length <= cutoff are returned.

Returns

lengths
[iterator] (source, shortest path length) iterator

See also:

single_source_shortest_path_length, **shortest_path_length**

Examples

```
>>> G = nx.path_graph(5, create_using=nx.DiGraph())
>>> length = dict(nx.single_target_shortest_path_length(G, 4))
>>> length[0]
4
>>> for node in range(5):
...     print(f"{node}: {length[node]}")
0: 4
1: 3
2: 2
3: 1
4: 0
```

bidirectional_shortest_path

bidirectional_shortest_path(*G*, *source*, *target*)

Returns a list of nodes in a shortest path between source and target.

Parameters

G
[NetworkX graph]

source
[node label] starting node for path

target
[node label] ending node for path

Returns

path: list
List of nodes in a path from source to target.

Raises

NetworkXNoPath
If no path exists between source and target.

See also:

shortest_path

Notes

This algorithm is used by shortest_path(G, source, target).

all_pairs_shortest_path

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

lengths
[dictionary] Dictionary, keyed by source and target, of shortest paths.

See also:

floyd_warshall

Examples

```
>>> G = nx.path_graph(5)
>>> path = dict(nx.all_pairs_shortest_path(G))
>>> print(path[0][4])
[0, 1, 2, 3, 4]
```

all_pairs_shortest_path_length

all_pairs_shortest_path_length(*G*, *cutoff=None*)

Computes the shortest path lengths between all nodes in `G`.

Parameters

G
[NetworkX graph]

cutoff
[integer, optional] Depth at which to stop the search. Only paths of length at most `cutoff` are returned.

Returns

lengths
[iterator] (source, dictionary) iterator with dictionary keyed by target and shortest path length as the key value.

Notes

The iterator returned only has reachable node pairs.

Examples

```
>>> G = nx.path_graph(5)
>>> length = dict(nx.all_pairs_shortest_path_length(G))
>>> for node in [0, 1, 2, 3, 4]:
...     print(f"1 - {node}: {length[1][node]}")
1 - 0: 1
1 - 1: 0
1 - 2: 1
1 - 3: 2
1 - 4: 3
>>> length[3][2]
1
>>> length[2][2]
0
```

predecessor

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})
```

Shortest path algorithms for weighted graphs.

dijkstra_predecessor_and_distance(G, source)	Compute weighted shortest path length and predecessors.
dijkstra_path(G, source, target[, weight])	Returns the shortest weighted path from source to target in G.
dijkstra_path_length(G, source, target[, weight])	Returns the shortest weighted path length in G from source to target.
single_source_dijkstra(G, source[, target, ...])	Find shortest weighted paths and lengths from a source node.
single_source_dijkstra_path(G, source[, ...])	Find shortest weighted paths in G from a source node.
single_source_dijkstra_path_length(G, source)	Find shortest weighted path lengths in G from a source node.
multi_source_dijkstra(G, sources[, target, ...])	Find shortest weighted paths and lengths from a given set of source nodes.
multi_source_dijkstra_path(G, sources[, ...])	Find shortest weighted paths in G from a given set of source nodes.
multi_source_dijkstra_path_length(G, sources)	Find shortest weighted path lengths in G from a given set of source nodes.
all_pairs_dijkstra(G[, cutoff, weight])	Find shortest weighted paths and lengths between all nodes.
all_pairs_dijkstra_path(G[, cutoff, weight])	Compute shortest paths between all nodes in a weighted graph.
all_pairs_dijkstra_path_length(G[, cutoff, ...])	Compute shortest path lengths between all nodes in a weighted graph.
bidirectional_dijkstra(G, source, target[, ...])	Dijkstra's algorithm for shortest paths using bidirectional search.
bellman_ford_path(G, source, target[, weight])	Returns the shortest path from source to target in a weighted graph G.
bellman_ford_path_length(G, source, target)	Returns the shortest path length from source to target in a weighted graph.
single_source_bellman_ford(G, source[, ...])	Compute shortest paths and lengths in a weighted graph G.
single_source_bellman_ford_path(G, source[, ...])	Compute shortest path between source and all other reachable nodes for a weighted graph.
single_source_bellman_ford_path_length(G, source)	Compute the shortest path length between source and all other reachable nodes for a weighted graph.
all_pairs_bellman_ford_path(G[, weight])	Compute shortest paths between all nodes in a weighted graph.
all_pairs_bellman_ford_path_length(G[, weight])	Compute shortest path lengths between all nodes in a weighted graph.
bellman_ford_predecessor_and_distance(G, source)	Compute shortest path lengths and predecessors on shortest paths in weighted graphs.
negative_edge_cycle(G[, weight, heuristic])	Returns True if there exists a negative edge cycle anywhere in G.
find_negative_cycle(G, source[, weight])	Returns a cycle with negative total weight if it exists.
goldberg_radzik(G, source[, weight])	Compute shortest path lengths and predecessors on shortest paths in weighted graphs.
johnson(G[, weight])	Uses Johnson's Algorithm to compute shortest paths.

dijkstra_predecessor_and_distance

dijkstra_predecessor_and_distance(*G*, *source*, *cutoff=None*, *weight='weight'*)

Compute weighted shortest path length and predecessors.

Uses Dijkstra's Method to obtain the shortest weighted paths and return dictionaries of predecessors for each node and distance for each node from the `source`.

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

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)]
```

dijkstra_path

dijkstra_path(*G*, *source*, *target*, *weight='weight'*)

Returns the shortest weighted path from source to target in G.

Uses Dijkstra's Method to compute the shortest weighted path between two nodes in a graph.

Parameters

G
[NetworkX graph]

source
[node] Starting node

target
[node] Ending node

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

path
[list] List of nodes in a shortest path.

Raises

NodeNotFound
If `source` is not in `G`.

NetworkXNoPath
If no path exists between source and target.

See also:

bidirectional_dijkstra
bellman_ford_path
single_source_dijkstra

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.

The weight function can be used to include node weights.

```
>>> def func(u, v, d):
...     node_u_wt = G.nodes[u].get("node_weight", 1)
...     node_v_wt = G.nodes[v].get("node_weight", 1)
...     edge_wt = d.get("weight", 1)
...     return node_u_wt / 2 + node_v_wt / 2 + edge_wt
```

In this example we take the average of start and end node weights of an edge and add it to the weight of the edge.

The function *single_source_dijkstra()* computes both path and length-of-path if you need both, use that.

Examples

```
>>> G = nx.path_graph(5)
>>> print(nx.dijkstra_path(G, 0, 4))
[0, 1, 2, 3, 4]
```

dijkstra_path_length

dijkstra_path_length(*G*, *source*, *target*, *weight='weight'*)

Returns the shortest weighted path length in G from source to target.

Uses Dijkstra's Method to compute the shortest weighted path length between two nodes in a graph.

Parameters

G
[NetworkX graph]

source
[node label] starting node for path

target
[node label] 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.

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
[number] Shortest path length.

Raises

NodeNotFound
If `source` is not in `G`.

NetworkXNoPath
If no path exists between source and target.

See also:

bidirectional_dijkstra
bellman_ford_path_length
single_source_dijkstra

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.

The function *single_source_dijkstra()* computes both path and length-of-path if you need both, use that.

Examples

```
>>> G = nx.path_graph(5)
>>> nx.dijkstra_path_length(G, 0, 4)
4
```

single_source_dijkstra

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 between a source and all other reachable nodes in a weighted graph.

Parameters

G
[NetworkX graph]

source
[node label] Starting node for path

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, paths and lengths to all nodes are computed. The return value is a tuple of two dictionaries keyed by target nodes. The first dictionary stores distance to each target node. The second stores the path to each target node. If target is not None, returns a tuple (distance, path), where distance is the distance from source to target and path is a list representing the path from source to target.

Raises

NodeNotFound
: If `source` is not in `G`.

See also:

single_source_dijkstra_path
single_source_dijkstra_path_length
single_source_bellman_ford

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 are negative or are floating point numbers (overflows and roundoff errors can cause problems).

Examples

```
>>> G = nx.path_graph(5)
>>> length, path = nx.single_source_dijkstra(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
>>> path[4]
[0, 1, 2, 3, 4]
```

(continues on next page)

(continued from previous page)

```
>>> length, path = nx.single_source_dijkstra(G, 0, 1)
>>> length
1
>>> path
[0, 1]
```

single_source_dijkstra_path

single_source_dijkstra_path(*G*, *source*, *cutoff=None*, *weight='weight'*)

Find shortest weighted paths in G from a source node.

Compute shortest path between source and all other reachable nodes for a weighted graph.

Parameters

G
[NetworkX graph]

source
[node] 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

paths
[dictionary] Dictionary of shortest path lengths keyed by target.

Raises

NodeNotFound
If `source` is not in `G`.

See also:

single_source_dijkstra*, *single_source_bellman_ford

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.

Examples

```
>>> G = nx.path_graph(5)
>>> path = nx.single_source_dijkstra_path(G, 0)
>>> path[4]
[0, 1, 2, 3, 4]
```

single_source_dijkstra_path_length

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`.

See also:

single_source_dijkstra, *single_source_bellman_ford_path_length*

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.

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
```

multi_source_dijkstra

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.

Raises

ValueError
: If `sources` is empty.

NodeNotFound
: If any of `sources` is not in `G`.

See also:

multi_source_dijkstra_path
multi_source_dijkstra_path_length

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 are negative or are floating point numbers (overflows and roundoff errors can cause problems).

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]
```

multi_source_dijkstra_path

multi_source_dijkstra_path(*G*, *sources*, *cutoff=None*, *weight='weight'*)

Find shortest weighted paths in G from a given set of source nodes.

Compute shortest path between any of the source nodes and all other reachable nodes 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.

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

paths
[dictionary] Dictionary of shortest paths keyed by target.

Raises

ValueError
If `sources` is empty.

NodeNotFound
If any of `sources` is not in `G`.

See also:

multi_source_dijkstra, **multi_source_bellman_ford**

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.

Examples

```
>>> G = nx.path_graph(5)
>>> path = nx.multi_source_dijkstra_path(G, {0, 4})
>>> path[1]
[0, 1]
>>> path[3]
[4, 3]
```

multi_source_dijkstra_path_length

multi_source_dijkstra_path_length(*G*, *sources*, *cutoff=None*, *weight='weight'*)

Find shortest weighted path lengths in G from a given set of source nodes.

Compute the shortest path length between any of the source nodes and all other reachable nodes 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.

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 to nearest source.

Raises

ValueError
If `sources` is empty.

NodeNotFound
If any of `sources` is not in `G`.

See also:

multi_source_dijkstra

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.

Examples

```
>>> G = nx.path_graph(5)
>>> length = nx.multi_source_dijkstra_path_length(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
```

all_pairs_dijkstra

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 dictionary of edge attributes for that edge. The function must return a number or None to indicate a hidden edge.

Yields

(node, (distance, path))
[(node obj, (dict, dict))] Each source node has two associated dicts. The first holds distance keyed by target and the second holds paths keyed by target. (See single_source_dijkstra for the source/target node terminology.) If desired you can apply `dict()` to this function to create a dict keyed by source node to the two dicts.

Notes

Edge weight attributes must be numerical. Distances are calculated as sums of weighted edges traversed.

The yielded dicts only have keys for reachable nodes.

Examples

```
>>> G = nx.path_graph(5)
>>> len_path = dict(nx.all_pairs_dijkstra(G))
>>> len_path[3][0][1]
2
>>> for node in [0, 1, 2, 3, 4]:
...     print(f"3 - {node}: {len_path[3][0][node]}")
3 - 0: 3
3 - 1: 2
3 - 2: 1
3 - 3: 0
3 - 4: 1
>>> len_path[3][1][1]
[3, 2, 1]
>>> for n, (dist, path) in nx.all_pairs_dijkstra(G):
...     print(path[1])
[0, 1]
[1]
[2, 1]
[3, 2, 1]
[4, 3, 2, 1]
```

all_pairs_dijkstra_path

all_pairs_dijkstra_path(*G*, *cutoff=None*, *weight='weight'*)

Compute shortest paths between all nodes in a weighted graph.

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.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
[dictionary] Dictionary, keyed by source and target, of shortest paths.

See also:

floyd_warshall, *all_pairs_bellman_ford_path*

Notes

Edge weight attributes must be numerical. Distances are calculated as sums of weighted edges traversed.

Examples

```
>>> G = nx.path_graph(5)
>>> path = dict(nx.all_pairs_dijkstra_path(G))
>>> path[0][4]
[0, 1, 2, 3, 4]
```

all_pairs_dijkstra_path_length

all_pairs_dijkstra_path_length(*G*, *cutoff=None*, *weight='weight'*)

Compute shortest path lengths between all nodes in a weighted graph.

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.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
[iterator] (source, dictionary) iterator with dictionary keyed by target and shortest path length as the key value.

Notes

Edge weight attributes must be numerical. Distances are calculated as sums of weighted edges traversed.

The dictionary returned only has keys for reachable node pairs.

Examples

```
>>> G = nx.path_graph(5)
>>> length = dict(nx.all_pairs_dijkstra_path_length(G))
>>> for node in [0, 1, 2, 3, 4]:
...     print(f"1 - {node}: {length[1][node]}")
1 - 0: 1
1 - 1: 0
1 - 2: 1
1 - 3: 2
1 - 4: 3
>>> length[3][2]
1
>>> length[2][2]
0
```

bidirectional_dijkstra

bidirectional_dijkstra(*G*, *source*, *target*, *weight='weight'*)

Dijkstra's algorithm for shortest paths using bidirectional search.

Parameters

G
[NetworkX graph]

source
[node] Starting node.

target
[node] Ending node.

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, path
[number and list] length is the distance from source to target. path is a list of nodes on a path from source to target.

Raises

NodeNotFound
If either `source` or `target` is not in `G`.

NetworkXNoPath
If no path exists between source and target.

See also:

shortest_path
shortest_path_length

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.

In practice bidirectional Dijkstra is much more than twice as fast as ordinary Dijkstra.

Ordinary Dijkstra expands nodes in a sphere-like manner from the source. The radius of this sphere will eventually be the length of the shortest path. Bidirectional Dijkstra will expand nodes from both the source and the target, making two spheres of half this radius. Volume of the first sphere is `pi*r*r` while the others are `2*pi*r/2*r/2`, making up half the volume.

This algorithm is not guaranteed to work if edge weights are negative or are floating point numbers (overflows and roundoff errors can cause problems).

Examples

```
>>> G = nx.path_graph(5)
>>> length, path = nx.bidirectional_dijkstra(G, 0, 4)
>>> print(length)
4
>>> print(path)
[0, 1, 2, 3, 4]
```

bellman_ford_path

bellman_ford_path(*G*, *source*, *target*, *weight='weight'*)

Returns the shortest path from source to target in a weighted graph G.

Parameters

G
[NetworkX graph]

source
[node] Starting node

target
[node] Ending node

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

path
[list] List of nodes in a shortest path.

Raises

NodeNotFound
If `source` is not in `G`.

NetworkXNoPath
If no path exists between source and target.

See also:

dijkstra_path*, *bellman_ford_path_length

Notes

Edge weight attributes must be numerical. Distances are calculated as sums of weighted edges traversed.

Examples

```
>>> G = nx.path_graph(5)
>>> nx.bellman_ford_path(G, 0, 4)
[0, 1, 2, 3, 4]
```

bellman_ford_path_length

bellman_ford_path_length(*G*, *source*, *target*, *weight='weight'*)

Returns the shortest path length from source to target in a weighted graph.

Parameters

G
[NetworkX graph]

source
[node label] starting node for path

target
[node label] ending 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
[number] Shortest path length.

Raises

NodeNotFound
If `source` is not in `G`.

NetworkXNoPath
If no path exists between source and target.

See also:

dijkstra_path_length*, *bellman_ford_path

Notes

Edge weight attributes must be numerical. Distances are calculated as sums of weighted edges traversed.

Examples

```
>>> G = nx.path_graph(5)
>>> nx.bellman_ford_path_length(G, 0, 4)
4
```

single_source_bellman_ford

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.

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

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.

Raises

NodeNotFound
: If `source` is not in `G`.

See also:

single_source_dijkstra
single_source_bellman_ford_path
single_source_bellman_ford_path_length

Notes

Edge weight attributes must be numerical. Distances are calculated as sums of weighted edges traversed.

Examples

```
>>> G = nx.path_graph(5)
>>> length, path = nx.single_source_bellman_ford(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
>>> path[4]
[0, 1, 2, 3, 4]
>>> length, path = nx.single_source_bellman_ford(G, 0, 1)
>>> length
1
>>> path
[0, 1]
```

single_source_bellman_ford_path

single_source_bellman_ford_path(*G*, *source*, *weight='weight'*)

Compute shortest path between source and all other reachable nodes for a weighted graph.

Parameters

G
: [NetworkX graph]

source
: [node] 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

paths
[dictionary] Dictionary of shortest path lengths keyed by target.

Raises

NodeNotFound
If `source` is not in `G`.

See also:

single_source_dijkstra*, *single_source_bellman_ford

Notes

Edge weight attributes must be numerical. Distances are calculated as sums of weighted edges traversed.

Examples

```
>>> G = nx.path_graph(5)
>>> path = nx.single_source_bellman_ford_path(G, 0)
>>> path[4]
[0, 1, 2, 3, 4]
```

single_source_bellman_ford_path_length

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
[iterator] (target, shortest path length) iterator

Raises

NodeNotFound
If `source` is not in `G`.

See also:

single_source_dijkstra*, *single_source_bellman_ford

Notes

Edge weight attributes must be numerical. Distances are calculated as sums of weighted edges traversed.

Examples

```
>>> G = nx.path_graph(5)
>>> length = dict(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
```

all_pairs_bellman_ford_path

all_pairs_bellman_ford_path(*G*, *weight='weight'*)

Compute shortest paths between all nodes in a weighted graph.

Parameters

G
[NetworkX graph]

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

distance
[dictionary] Dictionary, keyed by source and target, of shortest paths.

See also:

floyd_warshall, *all_pairs_dijkstra_path*

Notes

Edge weight attributes must be numerical. Distances are calculated as sums of weighted edges traversed.

Examples

```
>>> G = nx.path_graph(5)
>>> path = dict(nx.all_pairs_bellman_ford_path(G))
>>> path[0][4]
[0, 1, 2, 3, 4]
```

all_pairs_bellman_ford_path_length

all_pairs_bellman_ford_path_length(*G*, *weight='weight'*)

Compute shortest path lengths between all nodes in a weighted graph.

Parameters

G
[NetworkX graph]

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

distance
[iterator] (source, dictionary) iterator with dictionary keyed by target and shortest path length as the key value.

Notes

Edge weight attributes must be numerical. Distances are calculated as sums of weighted edges traversed.

The dictionary returned only has keys for reachable node pairs.

Examples

```
>>> G = nx.path_graph(5)
>>> length = dict(nx.all_pairs_bellman_ford_path_length(G))
>>> for node in [0, 1, 2, 3, 4]:
...     print(f"1 - {node}: {length[1][node]}")
1 - 0: 1
1 - 1: 0
1 - 2: 1
1 - 3: 2
1 - 4: 3
>>> length[3][2]
1
>>> length[2][2]
0
```

bellman_ford_predecessor_and_distance

bellman_ford_predecessor_and_distance(*G*, *source*, *target=None*, *weight='weight'*, *heuristic=False*)

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.

If a negative cycle is detected, you can use *find_negative_cycle()* to return the cycle and examine it. Shortest paths are not defined when a negative cycle exists because once reached, the path can cycle forever to build up arbitrarily low 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

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.

  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.

heuristic
: [bool] Determines whether to use a heuristic to early detect negative cycles at a hopefully negligible cost.

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.

See also:

find_negative_cycle

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 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 `[]`

Examples

```
>>> G = nx.path_graph(5, create_using=nx.DiGraph())
>>> pred, dist = nx.bellman_ford_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.bellman_ford_predecessor_and_distance(G, 0, 1)
>>> 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)]
```

```
>>> G = nx.cycle_graph(5, create_using=nx.DiGraph())
>>> G[1][2]["weight"] = -7
>>> nx.bellman_ford_predecessor_and_distance(G, 0)
Traceback (most recent call last):
    ...
networkx.exception.NetworkXUnbounded: Negative cycle detected.
```

negative_edge_cycle

negative_edge_cycle(*G*, *weight='weight'*, *heuristic=True*)

Returns True if there exists a negative edge cycle anywhere in G.

Parameters

G
[NetworkX graph]

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.

heuristic
[bool] Determines whether to use a heuristic to early detect negative cycles at a negligible cost. In case of graphs with a negative cycle, the performance of detection increases by at least an order of magnitude.

Returns

negative_cycle
[bool] True if a negative edge cycle exists, otherwise False.

Notes

Edge weight attributes must be numerical. Distances are calculated as sums of weighted edges traversed.

This algorithm uses bellman_ford_predecessor_and_distance() but finds negative cycles on any component by first adding a new node connected to every node, and starting bellman_ford_predecessor_and_distance on that node. It then removes that extra node.

Examples

```
>>> G = nx.cycle_graph(5, create_using=nx.DiGraph())
>>> print(nx.negative_edge_cycle(G))
False
>>> G[1][2]["weight"] = -7
>>> print(nx.negative_edge_cycle(G))
True
```

find_negative_cycle

find_negative_cycle(*G*, *source*, *weight='weight'*)

Returns a cycle with negative total weight if it exists.

Bellman-Ford is used to find shortest_paths. That algorithm stops if there exists a negative cycle. This algorithm picks up from there and returns the found negative cycle.

The cycle consists of a list of nodes in the cycle order. The last node equals the first to make it a cycle. You can look up the edge weights in the original graph. In the case of multigraphs the relevant edge is the minimal weight edge between the nodes in the 2-tuple.

If the graph has no negative cycle, a NetworkXError is raised.

Parameters

G
[NetworkX graph]

source: node label
The search for the negative cycle will start from this node.

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

cycle
[list] A list of nodes in the order of the cycle found. The last node equals the first to indicate a cycle.

Raises

NetworkXError
If no negative cycle is found.

Examples

```
>>> G = nx.DiGraph()
>>> G.add_weighted_edges_from([(0, 1, 2), (1, 2, 2), (2, 0, 1), (1, 4, 2), (4, 0,␣
↪-5)])
>>> nx.find_negative_cycle(G, 0)
[4, 0, 1, 4]
```

goldberg_radzik

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.

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 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.

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.
```

johnson

johnson(*G*, *weight='weight'*)

Uses Johnson's Algorithm to compute shortest paths.

Johnson's Algorithm finds a shortest path between each pair of nodes in a weighted graph even if negative weights are present.

Parameters

G
[NetworkX graph]

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

distance
[dictionary] Dictionary, keyed by source and target, of shortest paths.

Raises

NetworkXError
If given graph is not weighted.

See also:

floyd_warshall_predecessor_and_distance
floyd_warshall_numpy
all_pairs_shortest_path
all_pairs_shortest_path_length
all_pairs_dijkstra_path
bellman_ford_predecessor_and_distance
all_pairs_bellman_ford_path
all_pairs_bellman_ford_path_length

Notes

Johnson's algorithm is suitable even for graphs with negative weights. It works by using the Bellman–Ford algorithm to compute a transformation of the input graph that removes all negative weights, allowing Dijkstra's algorithm to be used on the transformed graph.

The time complexity of this algorithm is $O(n^2 \log n + nm)$, where n is the number of nodes and m the number of edges in the graph. For dense graphs, this may be faster than the Floyd–Warshall algorithm.

Examples

```
>>> graph = nx.DiGraph()
>>> graph.add_weighted_edges_from(
...     [("0", "3", 3), ("0", "1", -5), ("0", "2", 2), ("1", "2", 4), ("2", "3",␣
↪1)]
... )
>>> paths = nx.johnson(graph, weight="weight")
>>> paths["0"]["2"]
['0', '1', '2']
```

3.52.7 Dense Graphs

Floyd-Warshall algorithm for shortest paths.

floyd_warshall(G[, weight])	Find all-pairs shortest path lengths using Floyd's algorithm.
floyd_warshall_predecessor_and_distance(G[, ...])	Find all-pairs shortest path lengths using Floyd's algorithm.
floyd_warshall_numpy(G[, nodelist, weight])	Find all-pairs shortest path lengths using Floyd's algorithm.
reconstruct_path(source, target, predecessors)	Reconstruct a path from source to target using the predecessors dict as returned by floyd_warshall_predecessor_and_distance

floyd_warshall

floyd_warshall(*G*, *weight='weight'*)

Find all-pairs shortest path lengths using Floyd's algorithm.

Parameters

G
[NetworkX graph]

weight: string, optional (default= 'weight')
Edge data key corresponding to the edge weight.

Returns

distance
[dict] A dictionary, keyed by source and target, of shortest paths distances between nodes.

See also:

floyd_warshall_predecessor_and_distance
floyd_warshall_numpy
all_pairs_shortest_path
all_pairs_shortest_path_length

Notes

Floyd's algorithm is appropriate for finding shortest paths in dense graphs or graphs with negative weights when Dijkstra's algorithm fails. This algorithm can still fail if there are negative cycles. It has running time $O(n^3)$ with running space of $O(n^2)$.

floyd_warshall_predecessor_and_distance

floyd_warshall_predecessor_and_distance(*G*, *weight='weight'*)

Find all-pairs shortest path lengths using Floyd's algorithm.

Parameters

G
[NetworkX graph]

weight: string, optional (default= 'weight')
Edge data key corresponding to the edge weight.

Returns

predecessor,distance
[dictionaries] Dictionaries, keyed by source and target, of predecessors and distances in the shortest path.

See also:

floyd_warshall
floyd_warshall_numpy
all_pairs_shortest_path
all_pairs_shortest_path_length

Notes

Floyd's algorithm is appropriate for finding shortest paths in dense graphs or graphs with negative weights when Dijkstra's algorithm fails. This algorithm can still fail if there are negative cycles. It has running time $O(n^3)$ with running space of $O(n^2)$.

Examples

```
>>> G = nx.DiGraph()
>>> G.add_weighted_edges_from(
...     [
...         ("s", "u", 10),
...         ("s", "x", 5),
...         ("u", "v", 1),
...         ("u", "x", 2),
...         ("v", "y", 1),
```

(continues on next page)

(continued from previous page)

```
...           ("x", "u", 3),
...           ("x", "v", 5),
...           ("x", "y", 2),
...           ("y", "s", 7),
...           ("y", "v", 6),
...       ]
... )
>>> predecessors, _ = nx.floyd_warshall_predecessor_and_distance(G)
>>> print(nx.reconstruct_path("s", "v", predecessors))
['s', 'x', 'u', 'v']
```

floyd_warshall_numpy

floyd_warshall_numpy(*G*, *nodelist=None*, *weight='weight'*)

Find all-pairs shortest path lengths using Floyd's algorithm.

This algorithm for finding shortest paths takes advantage of matrix representations of a graph and works well for dense graphs where all-pairs shortest path lengths are desired. The results are returned as a NumPy array, distance[i, j], where i and j are the indexes of two nodes in nodelist. The entry distance[i, j] is the distance along a shortest path from i to j. If no path exists the distance is Inf.

Parameters

G
[NetworkX graph]

nodelist
[list, optional (default=G.nodes)] The rows and columns are ordered by the nodes in nodelist. If nodelist is None then the ordering is produced by G.nodes. Nodelist should include all nodes in G.

weight: string, optional (default='weight')
Edge data key corresponding to the edge weight.

Returns

distance
[2D numpy.ndarray] A numpy array of shortest path distances between nodes. If there is no path between two nodes the value is Inf.

Raises

NetworkXError
If nodelist is not a list of the nodes in G.

Notes

Floyd's algorithm is appropriate for finding shortest paths in dense graphs or graphs with negative weights when Dijkstra's algorithm fails. This algorithm can still fail if there are negative cycles. It has running time $O(n^3)$ with running space of $O(n^2)$.

reconstruct_path

reconstruct_path(*source*, *target*, *predecessors*)

Reconstruct a path from source to target using the predecessors dict as returned by floyd_warshall_predecessor_and_distance

Parameters

source
[node] Starting node for path

target
[node] Ending node for path

predecessors: dictionary
Dictionary, keyed by source and target, of predecessors in the shortest path, as returned by floyd_warshall_predecessor_and_distance

Returns

path
[list] A list of nodes containing the shortest path from source to target

If source and target are the same, an empty list is returned

See also:

floyd_warshall_predecessor_and_distance

Notes

This function is meant to give more applicability to the floyd_warshall_predecessor_and_distance function

3.52.8 A* Algorithm

Shortest paths and path lengths using the A* ("A star") algorithm.

astar_path(G, source, target[, heuristic, ...])	Returns a list of nodes in a shortest path between source and target using the A* ("A-star") algorithm.
astar_path_length(G, source, target[, ...])	Returns the length of the shortest path between source and target using the A* ("A-star") algorithm.

astar_path

astar_path(*G*, *source*, *target*, *heuristic=None*, *weight='weight'*)

Returns a list of nodes in a shortest path between source and target using the A* ("A-star") algorithm.

There may be more than one shortest path. This returns only one.

Parameters

G
[NetworkX graph]

source
[node] Starting node for path

target
: [node] Ending node for path

heuristic
: [function] A function to evaluate the estimate of the distance from the a node to the target. The function takes two nodes arguments and must return a number. If the heuristic is inadmissible (if it might overestimate the cost of reaching the goal from a node), the result may not be a shortest path. The algorithm does not support updating heuristic values for the same node due to caching the first heuristic calculation per node.

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.

Raises

NetworkXNoPath
: If no path exists between source and target.

See also:

`shortest_path`, `dijkstra_path`

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.

Examples

```
>>> G = nx.path_graph(5)
>>> print(nx.astar_path(G, 0, 4))
[0, 1, 2, 3, 4]
>>> G = nx.grid_graph(dim=[3, 3])  # nodes are two-tuples (x,y)
>>> nx.set_edge_attributes(G, {e: e[1][0] * 2 for e in G.edges()}, "cost")
>>> def dist(a, b):
...     (x1, y1) = a
...     (x2, y2) = b
...     return ((x1 - x2) ** 2 + (y1 - y2) ** 2) ** 0.5
>>> print(nx.astar_path(G, (0, 0), (2, 2), heuristic=dist, weight="cost"))
[(0, 0), (0, 1), (0, 2), (1, 2), (2, 2)]
```

astar_path_length

astar_path_length(*G*, *source*, *target*, *heuristic=None*, *weight='weight'*)

Returns the length of the shortest path between source and target using the A* ("A-star") algorithm.

Parameters

G
[NetworkX graph]

source
[node] Starting node for path

target
[node] Ending node for path

heuristic
[function] A function to evaluate the estimate of the distance from the a node to the target. The function takes two nodes arguments and must return a number. If the heuristic is inadmissible (if it might overestimate the cost of reaching the goal from a node), the result may not be a shortest path. The algorithm does not support updating heuristic values for the same node due to caching the first heuristic calculation per node.

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.

Raises

NetworkXNoPath
If no path exists between source and target.

See also:

astar_path

3.53 Similarity Measures

Functions measuring similarity using graph edit distance.

The graph edit distance is the number of edge/node changes needed to make two graphs isomorphic.

The default algorithm/implementation is sub-optimal for some graphs. The problem of finding the exact Graph Edit Distance (GED) is NP-hard so it is often slow. If the simple interface *graph_edit_distance* takes too long for your graph, try *optimize_graph_edit_distance* and/or *optimize_edit_paths*.

At the same time, I encourage capable people to investigate alternative GED algorithms, in order to improve the choices available.

graph_edit_distance(G1, G2[, node_match, ...])	Returns GED (graph edit distance) between graphs G1 and G2.
optimal_edit_paths(G1, G2[, node_match, ...])	Returns all minimum-cost edit paths transforming G1 to G2.
optimize_graph_edit_distance(G1, G2[, ...])	Returns consecutive approximations of GED (graph edit distance) between graphs G1 and G2.
optimize_edit_paths(G1, G2[, node_match, ...])	GED (graph edit distance) calculation: advanced interface.
simrank_similarity(G[, source, target, ...])	Returns the SimRank similarity of nodes in the graph `G`.
panther_similarity(G, source[, k, ...])	Returns the Panther similarity of nodes in the graph `G` to node `v`.
generate_random_paths(G, sample_size[, ...])	Randomly generate `sample_size` paths of length `path_length`.

3.53.1 graph_edit_distance

graph_edit_distance(*G1*, *G2*, *node_match=None*, *edge_match=None*, *node_subst_cost=None*, *node_del_cost=None*, *node_ins_cost=None*, *edge_subst_cost=None*, *edge_del_cost=None*, *edge_ins_cost=None*, *roots=None*, *upper_bound=None*, *timeout=None*)

Returns GED (graph edit distance) between graphs G1 and G2.

Graph edit distance is a graph similarity measure analogous to Levenshtein distance for strings. It is defined as minimum cost of edit path (sequence of node and edge edit operations) transforming graph G1 to graph isomorphic to G2.

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.

See also:

optimal_edit_paths, *optimize_graph_edit_distance*

is_isomorphic
test for graph edit distance of 0

References

[1]

Examples

```
>>> G1 = nx.cycle_graph(6)
>>> G2 = nx.wheel_graph(7)
>>> nx.graph_edit_distance(G1, G2)
7.0
```

```
>>> G1 = nx.star_graph(5)
>>> G2 = nx.star_graph(5)
>>> nx.graph_edit_distance(G1, G2, roots=(0, 0))
0.0
>>> nx.graph_edit_distance(G1, G2, roots=(1, 0))
8.0
```

3.53.2 optimal_edit_paths

optimal_edit_paths(*G1*, *G2*, *node_match=None*, *edge_match=None*, *node_subst_cost=None*, *node_del_cost=None*, *node_ins_cost=None*, *edge_subst_cost=None*, *edge_del_cost=None*, *edge_ins_cost=None*, *upper_bound=None*)

Returns all minimum-cost edit paths transforming G1 to G2.

Graph edit path is a sequence of node and edge edit operations transforming graph G1 to graph isomorphic to G2. Edit operations include substitutions, deletions, and insertions.

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.

upper_bound
[numeric] Maximum edit distance to consider.

Returns

edit_paths
[list of tuples (node_edit_path, edge_edit_path)] node_edit_path : list of tuples (u, v) edge_edit_path : list of tuples ((u1, v1), (u2, v2))

cost
[numeric] Optimal edit path cost (graph edit distance).

See also:

graph_edit_distance, *optimize_edit_paths*

References

[1]

Examples

```
>>> G1 = nx.cycle_graph(4)
>>> G2 = nx.wheel_graph(5)
>>> paths, cost = nx.optimal_edit_paths(G1, G2)
>>> len(paths)
40
>>> cost
5.0
```

3.53.3 optimize_graph_edit_distance

optimize_graph_edit_distance(*G1*, *G2*, *node_match=None*, *edge_match=None*, *node_subst_cost=None*, *node_del_cost=None*, *node_ins_cost=None*, *edge_subst_cost=None*, *edge_del_cost=None*, *edge_ins_cost=None*, *upper_bound=None*)

Returns consecutive approximations of GED (graph edit distance) between graphs G1 and G2.

Graph edit distance is a graph similarity measure analogous to Levenshtein distance for strings. It is defined as minimum cost of edit path (sequence of node and edge edit operations) transforming graph G1 to graph isomorphic to G2.

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.

upper_bound
[numeric] Maximum edit distance to consider.

Returns

Generator of consecutive approximations of graph edit distance.

See also:

graph_edit_distance, *optimize_edit_paths*

References

[1]

Examples

```
>>> G1 = nx.cycle_graph(6)
>>> G2 = nx.wheel_graph(7)
>>> for v in nx.optimize_graph_edit_distance(G1, G2):
...     minv = v
>>> minv
7.0
```

3.53.4 optimize_edit_paths

optimize_edit_paths(*G1*, *G2*, *node_match=None*, *edge_match=None*, *node_subst_cost=None*, *node_del_cost=None*, *node_ins_cost=None*, *edge_subst_cost=None*, *edge_del_cost=None*, *edge_ins_cost=None*, *upper_bound=None*, *strictly_decreasing=True*, *roots=None*, *timeout=None*)

GED (graph edit distance) calculation: advanced interface.

Graph edit path is a sequence of node and edge edit operations transforming graph G1 to graph isomorphic to G2. Edit operations include substitutions, deletions, and insertions.

Graph edit distance is defined as minimum cost of edit path.

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.

upper_bound
[numeric] Maximum edit distance to consider.

strictly_decreasing
[bool] If True, return consecutive approximations of strictly decreasing cost. Otherwise, return all edit paths of cost less than or equal to the previous minimum cost.

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.

timeout
[numeric] Maximum number of seconds to execute. After timeout is met, the current best GED is returned.

Returns

Generator of tuples (node_edit_path, edge_edit_path, cost)
node_edit_path : list of tuples (u, v) edge_edit_path : list of tuples ((u1, v1), (u2, v2)) cost : numeric

See also:

graph_edit_distance, *optimize_graph_edit_distance*, *optimal_edit_paths*

References

[1]

3.53.5 simrank_similarity

simrank_similarity(*G*, *source=None*, *target=None*, *importance_factor=0.9*, *max_iterations=1000*, *tolerance=0.0001*)

Returns the SimRank similarity of nodes in the graph `G`.

SimRank is a similarity metric that says "two objects are considered to be similar if they are referenced by similar objects." [1].

The pseudo-code definition from the paper is:

```
def simrank(G, u, v):
    in_neighbors_u = G.predecessors(u)
    in_neighbors_v = G.predecessors(v)
    scale = C / (len(in_neighbors_u) * len(in_neighbors_v))
    return scale * sum(simrank(G, w, x)
                       for w, x in product(in_neighbors_u,
                                           in_neighbors_v))
```

where `G` is the graph, `u` is the source, `v` is the target, and `C` is a float decay or importance factor between 0 and 1.

The SimRank algorithm for determining node similarity is defined in [2].

Parameters

G
[NetworkX graph] A NetworkX graph

source
[node] If this is specified, the returned dictionary maps each node `v` in the graph to the similarity between `source` and `v`.

target
[node] If both `source` and `target` are specified, the similarity value between `source` and `target` is returned. If `target` is specified but `source` is not, this argument is ignored.

importance_factor
[float] The relative importance of indirect neighbors with respect to direct neighbors.

max_iterations
[integer] 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
[dictionary or float] If `source` and `target` are both `None`, this returns a dictionary of dictionaries, where keys are node pairs and value are similarity of the pair of nodes.

If `source` is not `None` but `target` is, this returns a dictionary mapping node to the similarity of `source` and that node.

If neither `source` nor `target` is `None`, this returns the similarity value for the given pair of nodes.

References

[1], [2]

Examples

```
>>> G = nx.cycle_graph(2)
>>> nx.simrank_similarity(G)
{0: {0: 1.0, 1: 0.0}, 1: {0: 0.0, 1: 1.0}}
>>> nx.simrank_similarity(G, source=0)
{0: 1.0, 1: 0.0}
>>> nx.simrank_similarity(G, source=0, target=0)
1.0
```

The result of this function can be converted to a numpy array representing the SimRank matrix by using the node order of the graph to determine which row and column represent each node. Other ordering of nodes is also possible.

```
>>> import numpy as np
>>> sim = nx.simrank_similarity(G)
>>> np.array([[sim[u][v] for v in G] for u in G])
array([[1., 0.],
       [0., 1.]])
>>> sim_1d = nx.simrank_similarity(G, source=0)
>>> np.array([sim[0][v] for v in G])
array([1., 0.])
```

3.53.6 panther_similarity

panther_similarity(*G*, *source*, *k=5*, *path_length=5*, *c=0.5*, *delta=0.1*, *eps=None*)

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 ϕ 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.

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.

References

[1]

Examples

```
>>> G = nx.star_graph(10)
>>> sim = nx.panther_similarity(G, 0)
```

3.53.7 generate_random_paths

generate_random_paths(*G*, *sample_size*, *path_length=5*, *index_map=None*)

Randomly generate `sample_size` paths of length `path_length`.

Parameters

G
[NetworkX graph] A NetworkX graph

sample_size
[integer] The number of paths to generate. This is `R` in [1].

path_length
[integer (default = 5)] The maximum size of the path to randomly generate. This is `T` in [1]. According to the paper, `T >= 5` is recommended.

index_map
[dictionary, optional] If provided, this will be populated with the inverted index of nodes mapped to the set of generated random path indices within `paths`.

Returns

paths
[generator of lists] Generator of `sample_size` paths each with length `path_length`.

References

[1]

Examples

Note that the return value is the list of paths:

```
>>> G = nx.star_graph(3)
>>> random_path = nx.generate_random_paths(G, 2)
```

By passing a dictionary into `index_map`, it will build an inverted index mapping of nodes to the paths in which that node is present:

```
>>> G = nx.star_graph(3)
>>> index_map = {}
>>> random_path = nx.generate_random_paths(G, 3, index_map=index_map)
>>> paths_containing_node_0 = [random_path[path_idx] for path_idx in index_map.
↪get(0, [])]
```

3.54 Simple Paths

all_simple_paths(G, source, target[, cutoff])	Generate all simple paths in the graph G from source to target.
all_simple_edge_paths(G, source, target[, ...])	Generate lists of edges for all simple paths in G from source to target.
is_simple_path(G, nodes)	Returns True if and only if `nodes` form a simple path in `G`.
shortest_simple_paths(G, source, target[, ...])	Generate all simple paths in the graph G from source to target,

3.54.1 all_simple_paths

all_simple_paths(*G*, *source*, *target*, *cutoff=None*)

Generate all simple paths in the graph G from source to target.

A simple path is a path with no repeated nodes.

Parameters

G
[NetworkX graph]

source
[node] Starting node for path

target
[nodes] Single node or iterable of nodes at which to end path

cutoff
[integer, optional] Depth to stop the search. Only paths of length <= cutoff are returned.

Returns

path_generator: generator
A generator that produces lists of simple paths. If there are no paths between the source and target within the given cutoff the generator produces no output. If it is possible to traverse the same sequence of nodes in multiple ways, namely through parallel edges, then it will be returned multiple times (once for each viable edge combination).

See also:

all_shortest_paths, shortest_path, has_path

Notes

This algorithm uses a modified depth-first search to generate the paths [1]. A single path can be found in $O(V+E)$ time but the number of simple paths in a graph can be very large, e.g. $O(n!)$ in the complete graph of order n.

This function does not check that a path exists between `source` and `target`. For large graphs, this may result in very long runtimes. Consider using `has_path` to check that a path exists between `source` and `target` before calling this function on large graphs.

References

[1]

Examples

This iterator generates lists of nodes:

```
>>> G = nx.complete_graph(4)
>>> for path in nx.all_simple_paths(G, source=0, target=3):
...     print(path)
...
[0, 1, 2, 3]
[0, 1, 3]
[0, 2, 1, 3]
[0, 2, 3]
[0, 3]
```

You can generate only those paths that are shorter than a certain length by using the `cutoff` keyword argument:

```
>>> paths = nx.all_simple_paths(G, source=0, target=3, cutoff=2)
>>> print(list(paths))
[[0, 1, 3], [0, 2, 3], [0, 3]]
```

To get each path as the corresponding list of edges, you can use the `networkx.utils.pairwise()` helper function:

```
>>> paths = nx.all_simple_paths(G, source=0, target=3)
>>> for path in map(nx.utils.pairwise, paths):
...     print(list(path))
[(0, 1), (1, 2), (2, 3)]
[(0, 1), (1, 3)]
[(0, 2), (2, 1), (1, 3)]
[(0, 2), (2, 3)]
[(0, 3)]
```

Pass an iterable of nodes as target to generate all paths ending in any of several nodes:

```
>>> G = nx.complete_graph(4)
>>> for path in nx.all_simple_paths(G, source=0, target=[3, 2]):
...     print(path)
...
[0, 1, 2]
[0, 1, 2, 3]
[0, 1, 3]
[0, 1, 3, 2]
[0, 2]
[0, 2, 1, 3]
[0, 2, 3]
[0, 3]
[0, 3, 1, 2]
[0, 3, 2]
```

Iterate over each path from the root nodes to the leaf nodes in a directed acyclic graph using a functional programming approach:

```
>>> from itertools import chain
>>> from itertools import product
>>> from itertools import starmap
>>> from functools import partial
>>>
>>> chaini = chain.from_iterable
>>>
>>> G = nx.DiGraph([(0, 1), (1, 2), (0, 3), (3, 2)])
>>> roots = (v for v, d in G.in_degree() if d == 0)
>>> leaves = (v for v, d in G.out_degree() if d == 0)
>>> all_paths = partial(nx.all_simple_paths, G)
>>> list(chaini(starmap(all_paths, product(roots, leaves))))
[[0, 1, 2], [0, 3, 2]]
```

The same list computed using an iterative approach:

```
>>> G = nx.DiGraph([(0, 1), (1, 2), (0, 3), (3, 2)])
>>> roots = (v for v, d in G.in_degree() if d == 0)
>>> leaves = (v for v, d in G.out_degree() if d == 0)
>>> all_paths = []
>>> for root in roots:
...     for leaf in leaves:
...         paths = nx.all_simple_paths(G, root, leaf)
...         all_paths.extend(paths)
>>> all_paths
[[0, 1, 2], [0, 3, 2]]
```

Iterate over each path from the root nodes to the leaf nodes in a directed acyclic graph passing all leaves together to avoid unnecessary compute:

```
>>> G = nx.DiGraph([(0, 1), (2, 1), (1, 3), (1, 4)])
>>> roots = (v for v, d in G.in_degree() if d == 0)
>>> leaves = [v for v, d in G.out_degree() if d == 0]
>>> all_paths = []
>>> for root in roots:
...     paths = nx.all_simple_paths(G, root, leaves)
...     all_paths.extend(paths)
```

(continues on next page)

(continued from previous page)

```
>>> all_paths
[[0, 1, 3], [0, 1, 4], [2, 1, 3], [2, 1, 4]]
```

If parallel edges offer multiple ways to traverse a given sequence of nodes, this sequence of nodes will be returned multiple times:

```
>>> G = nx.MultiDiGraph([(0, 1), (0, 1), (1, 2)])
>>> list(nx.all_simple_paths(G, 0, 2))
[[0, 1, 2], [0, 1, 2]]
```

3.54.2 all_simple_edge_paths

all_simple_edge_paths(*G*, *source*, *target*, *cutoff=None*)

Generate lists of edges for all simple paths in G from source to target.

A simple path is a path with no repeated nodes.

Parameters

G
[NetworkX graph]

source
[node] Starting node for path

target
[nodes] Single node or iterable of nodes at which to end path

cutoff
[integer, optional] Depth to stop the search. Only paths of length <= cutoff are returned.

Returns

path_generator: generator
A generator that produces lists of simple paths. If there are no paths between the source and target within the given cutoff the generator produces no output. For multigraphs, the list of edges have elements of the form `(u,v,k)`. Where `k` corresponds to the edge key.

See also:

all_shortest_paths, shortest_path, *all_simple_paths*

Notes

This algorithm uses a modified depth-first search to generate the paths [1]. A single path can be found in $O(V+E)$ time but the number of simple paths in a graph can be very large, e.g. $O(n!)$ in the complete graph of order n.

References

[1]

Examples

Print the simple path edges of a Graph:

```
>>> g = nx.Graph([(1, 2), (2, 4), (1, 3), (3, 4)])
>>> for path in sorted(nx.all_simple_edge_paths(g, 1, 4)):
...     print(path)
[(1, 2), (2, 4)]
[(1, 3), (3, 4)]
```

Print the simple path edges of a MultiGraph. Returned edges come with their associated keys:

```
>>> mg = nx.MultiGraph()
>>> mg.add_edge(1, 2, key="k0")
'k0'
>>> mg.add_edge(1, 2, key="k1")
'k1'
>>> mg.add_edge(2, 3, key="k0")
'k0'
>>> for path in sorted(nx.all_simple_edge_paths(mg, 1, 3)):
...     print(path)
[(1, 2, 'k0'), (2, 3, 'k0')]
[(1, 2, 'k1'), (2, 3, 'k0')]
```

3.54.3 is_simple_path

is_simple_path(*G*, *nodes*)

Returns True if and only if `nodes` form a simple path in `G`.

A *simple path* in a graph is a nonempty sequence of nodes in which no node appears more than once in the sequence, and each adjacent pair of nodes in the sequence is adjacent in the graph.

Parameters

G
[graph] A NetworkX graph.

nodes
[list] A list of one or more nodes in the graph `G`.

Returns

bool
Whether the given list of nodes represents a simple path in `G`.

Notes

An empty list of nodes is not a path but a list of one node is a path. Here's an explanation why.

This function operates on *node paths*. One could also consider *edge paths*. There is a bijection between node paths and edge paths.

The *length of a path* is the number of edges in the path, so a list of nodes of length *n* corresponds to a path of length *n* - 1. Thus the smallest edge path would be a list of zero edges, the empty path. This corresponds to a list of one node.

To convert between a node path and an edge path, you can use code like the following:

```
>>> from networkx.utils import pairwise
>>> nodes = [0, 1, 2, 3]
>>> edges = list(pairwise(nodes))
>>> edges
[(0, 1), (1, 2), (2, 3)]
>>> nodes = [edges[0][0]] + [v for u, v in edges]
>>> nodes
[0, 1, 2, 3]
```

Examples

```
>>> G = nx.cycle_graph(4)
>>> nx.is_simple_path(G, [2, 3, 0])
True
>>> nx.is_simple_path(G, [0, 2])
False
```

3.54.4 shortest_simple_paths

shortest_simple_paths(*G*, *source*, *target*, *weight=None*)

Generate all simple paths in the graph G from source to target,
starting from shortest ones.

A simple path is a path with no repeated nodes.

If a weighted shortest path search is to be used, no negative weights are allowed.

Parameters

G
[NetworkX graph]

source
[node] Starting node for path

target
[node] Ending node for path

weight
[string or function] If it is a string, it is the name of the edge attribute to be used as a weight.

If it 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.

If None all edges are considered to have unit weight. Default value None.

Returns

path_generator: generator
A generator that produces lists of simple paths, in order from shortest to longest.

Raises

NetworkXNoPath
If no path exists between source and target.

NetworkXError
If source or target nodes are not in the input graph.

NetworkXNotImplemented
If the input graph is a Multi[Di]Graph.

See also:

all_shortest_paths
shortest_path
all_simple_paths

Notes

This procedure is based on algorithm by Jin Y. Yen [1]. Finding the first K paths requires $O(KN^3)$ operations.

References

[1]

Examples

```
>>> G = nx.cycle_graph(7)
>>> paths = list(nx.shortest_simple_paths(G, 0, 3))
>>> print(paths)
[[0, 1, 2, 3], [0, 6, 5, 4, 3]]
```

You can use this function to efficiently compute the k shortest/best paths between two nodes.

```
>>> from itertools import islice
>>> def k_shortest_paths(G, source, target, k, weight=None):
...     return list(
...         islice(nx.shortest_simple_paths(G, source, target, weight=weight), k)
...     )
>>> for path in k_shortest_paths(G, 0, 3, 2):
...     print(path)
[0, 1, 2, 3]
[0, 6, 5, 4, 3]
```

3.55 Small-world

Functions for estimating the small-world-ness of graphs.

A small world network is characterized by a small average shortest path length, and a large clustering coefficient.

Small-worldness is commonly measured with the coefficient sigma or omega.

Both coefficients compare the average clustering coefficient and shortest path length of a given graph against the same quantities for an equivalent random or lattice graph.

For more information, see the Wikipedia article on small-world network [1].

random_reference(G[, niter, connectivity, seed])	Compute a random graph by swapping edges of a given graph.
lattice_reference(G[, niter, D, ...])	Latticize the given graph by swapping edges.
sigma(G[, niter, nrand, seed])	Returns the small-world coefficient (sigma) of the given graph.
omega(G[, niter, nrand, seed])	Returns the small-world coefficient (omega) of a graph

3.55.1 random_reference

random_reference(*G*, *niter=1*, *connectivity=True*, *seed=None*)

Compute a random graph by swapping edges of a given graph.

Parameters

G
[graph] An undirected graph with 4 or more nodes.

niter
[integer (optional, default=1)] An edge is rewired approximately `niter` times.

connectivity
[boolean (optional, default=True)] When True, ensure connectivity for the randomized graph.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

G
[graph] The randomized graph.

Raises

NetworkXError
If there are fewer than 4 nodes or 2 edges in `G`

Notes

The implementation is adapted from the algorithm by Maslov and Sneppen (2002) [1].

References

[1]

3.55.2 lattice_reference

lattice_reference(*G*, *niter=5*, *D=None*, *connectivity=True*, *seed=None*)

Latticize the given graph by swapping edges.

Parameters

G
[graph] An undirected graph.

niter
[integer (optional, default=1)] An edge is rewired approximatively niter times.

D
[numpy.array (optional, default=None)] Distance to the diagonal matrix.

connectivity
[boolean (optional, default=True)] Ensure connectivity for the latticized graph when set to True.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

G
[graph] The latticized graph.

Raises

NetworkXError
If there are fewer than 4 nodes or 2 edges in `G`

Notes

The implementation is adapted from the algorithm by Sporns et al. [1]. which is inspired from the original work by Maslov and Sneppen(2002) [2].

References

[1], [2]

3.55.3 sigma

sigma(*G*, *niter=100*, *nrand=10*, *seed=None*)

Returns the small-world coefficient (sigma) of the given graph.

The small-world coefficient is defined as: sigma = C/Cr / L/Lr where C and L are respectively the average clustering coefficient and average shortest path length of G. Cr and Lr are respectively the average clustering coefficient and average shortest path length of an equivalent random graph.

A graph is commonly classified as small-world if sigma>1.

Parameters

G
[NetworkX graph] An undirected graph.

niter
[integer (optional, default=100)] Approximate number of rewiring per edge to compute the equivalent random graph.

nrand
[integer (optional, default=10)] Number of random graphs generated to compute the average clustering coefficient (Cr) and average shortest path length (Lr).

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

sigma
[float] The small-world coefficient of G.

Notes

The implementation is adapted from Humphries et al. [1] [2].

References

[1], [2]

3.55.4 omega

omega(*G*, *niter=5*, *nrand=10*, *seed=None*)

Returns the small-world coefficient (omega) of a graph

The small-world coefficient of a graph G is:

omega = Lr/L - C/Cl

where C and L are respectively the average clustering coefficient and average shortest path length of G. Lr is the average shortest path length of an equivalent random graph and Cl is the average clustering coefficient of an equivalent lattice graph.

The small-world coefficient (omega) measures how much G is like a lattice or a random graph. Negative values mean G is similar to a lattice whereas positive values mean G is a random graph. Values close to 0 mean that G has small-world characteristics.

Parameters

G
[NetworkX graph] An undirected graph.

niter: integer (optional, default=5)
Approximate number of rewiring per edge to compute the equivalent random graph.

nrand: integer (optional, default=10)
Number of random graphs generated to compute the maximal clustering coefficient (Cr) and average shortest path length (Lr).

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

omega
[float] The small-world coefficient (omega)

Notes

The implementation is adapted from the algorithm by Telesford et al. [1].

References

[1]

3.56 s metric

s_metric(G[, normalized])	Returns the s-metric of graph.

3.56.1 s_metric

s_metric(*G*, *normalized=True*)

Returns the s-metric of graph.

The s-metric is defined as the sum of the products deg(u)*deg(v) for every edge (u,v) in G. If norm is provided construct the s-max graph and compute it's s_metric, and return the normalized s value

Parameters

G
[graph] The graph used to compute the s-metric.

normalized
[bool (optional)] Normalize the value.

Returns

s
[float] The s-metric of the graph.

References

[1]

3.57 Sparsifiers

Functions for computing sparsifiers of graphs.

spanner(G, stretch[, weight, seed])	Returns a spanner of the given graph with the given stretch.

3.57.1 spanner

spanner(*G*, *stretch*, *weight=None*, *seed=None*)

Returns a spanner of the given graph with the given stretch.

A spanner of a graph G = (V, E) with stretch t is a subgraph H = (V, E_S) such that E_S is a subset of E and the distance between any pair of nodes in H is at most t times the distance between the nodes in G.

Parameters

G
[NetworkX graph] An undirected simple graph.

stretch
[float] The stretch of the spanner.

weight
[object] The edge attribute to use as distance.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

NetworkX graph
A spanner of the given graph with the given stretch.

Raises

ValueError
If a stretch less than 1 is given.

Notes

This function implements the spanner algorithm by Baswana and Sen, see [1].

This algorithm is a randomized las vegas algorithm: The expected running time is O(km) where k = (stretch + 1) // 2 and m is the number of edges in G. The returned graph is always a spanner of the given graph with the specified stretch. For weighted graphs the number of edges in the spanner is O(k * n^(1 + 1 / k)) where k is defined as above and n is the number of nodes in G. For unweighted graphs the number of edges is O(n^(1 + 1 / k) + kn).

References

[1] S. Baswana, S. Sen. A Simple and Linear Time Randomized Algorithm for Computing Sparse Spanners in Weighted Graphs. Random Struct. Algorithms 30(4): 532-563 (2007).

3.58 Structural holes

Functions for computing measures of structural holes.

constraint(G[, nodes, weight])	Returns the constraint on all nodes in the graph G.
effective_size(G[, nodes, weight])	Returns the effective size of all nodes in the graph G.
local_constraint(G, u, v[, weight])	Returns the local constraint on the node u with respect to the node v in the graph G.

3.58.1 constraint

constraint(*G*, *nodes=None*, *weight=None*)

Returns the constraint on all nodes in the graph G.

The *constraint* is a measure of the extent to which a node *v* is invested in those nodes that are themselves invested in the neighbors of *v*. Formally, the *constraint on v*, denoted c(v), is defined by

$$c(v) = \sum_{w \in N(v) \setminus \{v\}} \ell(v, w)$$

where $N(v)$ is the subset of the neighbors of v that are either predecessors or successors of v and $\ell(v, w)$ is the local constraint on v with respect to w [1]. For the definition of local constraint, see *local_constraint()*.

Parameters

G
[NetworkX graph] The graph containing v. This can be either directed or undirected.

nodes
[container, optional] Container of nodes in the graph G to compute the constraint. If None, the constraint 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 constraint on the node as values.

See also:

local_constraint

References

[1]

3.58.2 effective_size

effective_size(*G*, *nodes=None*, *weight=None*)

Returns the effective size of all nodes in the graph `G`.

The *effective size* of a node's ego network is based on the concept of redundancy. A person's ego network has redundancy to the extent that her contacts are connected to each other as well. The nonredundant part of a person's relationships it's the effective size of her ego network [1]. Formally, the effective size of a node u, denoted $e(u)$, is defined by

$$e(u) = \sum_{v \in N(u) \setminus \{u\}} \left(1 - \sum_{w \in N(v)} p_{uw} m_{vw}\right)$$

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]

$$e(u) = n - \frac{2t}{n}$$

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.

See also:

constraint

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()}
```

References

[1], [2]

3.58.3 local_constraint

local_constraint(*G*, *u*, *v*, *weight=None*)

Returns the local constraint on the node `u` with respect to the node `v` in the graph `G`.

Formally, the *local constraint on u with respect to v*, denoted $\ell(v)$, is defined by

$$\ell(u, v) = \left(p_{uv} + \sum_{w \in N(v)} p_{uw} p_{wv}\right)^2,$$

where $N(v)$ is the set of neighbors of v and p_{uv} is the normalized mutual weight of the (directed or undirected) edges joining u and v, for each vertex u and v [1]. 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).

Parameters

G
: [NetworkX graph] The graph containing `u` and `v`. This can be either directed or undirected.

u
: [node] A node in the graph `G`.

v
: [node] A node in the graph `G`.

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

float
: The constraint of the node `v` in the graph `G`.

See also:

constraint

References

[1]

3.59 Summarization

Graph summarization finds smaller representations of graphs resulting in faster runtime of algorithms, reduced storage needs, and noise reduction. Summarization has applications in areas such as visualization, pattern mining, clustering and community detection, and more. Core graph summarization techniques are grouping/aggregation, bit-compression, simplification/sparsification, and influence based. Graph summarization algorithms often produce either summary graphs in the form of supergraphs or sparsified graphs, or a list of independent structures. Supergraphs are the most common product, which consist of supernodes and original nodes and are connected by edges and superedges, which represent aggregate edges between nodes and supernodes.

Grouping/aggregation based techniques compress graphs by representing close/connected nodes and edges in a graph by a single node/edge in a supergraph. Nodes can be grouped together into supernodes based on their structural similarities or proximity within a graph to reduce the total number of nodes in a graph. Edge-grouping techniques group edges into lossy/lossless nodes called compressor or virtual nodes to reduce the total number of edges in a graph. Edge-grouping techniques can be lossless, meaning that they can be used to re-create the original graph, or techniques can be lossy, requiring less space to store the summary graph, but at the expense of lower recontruction accuracy of the original graph.

Bit-compression techniques minimize the amount of information needed to describe the original graph, while revealing structural patterns in the original graph. The two-part minimum description length (MDL) is often used to represent the model and the original graph in terms of the model. A key difference between graph compression and graph summarization is that graph summarization focuses on finding structural patterns within the original graph, whereas graph compression focuses on compressions the original graph to be as small as possible. **NOTE**: Some bit-compression methods exist solely to compress a graph without creating a summary graph or finding comprehensible structural patterns.

Simplification/Sparsification techniques attempt to create a sparse representation of a graph by removing unimportant nodes and edges from the graph. Sparsified graphs differ from supergraphs created by grouping/aggregation by only containing a subset of the original nodes and edges of the original graph.

Influence based techniques aim to find a high-level description of influence propagation in a large graph. These methods are scarce and have been mostly applied to social graphs.

dedensification is a grouping/aggregation based technique to compress the neighborhoods around high-degree nodes in unweighted graphs by adding compressor nodes that summarize multiple edges of the same type to high-degree nodes (nodes with a degree greater than a given threshold). Dedensification was developed for the purpose of increasing performance of query processing around high-degree nodes in graph databases and enables direct operations on the compressed graph. The structural patterns surrounding high-degree nodes in the original is preserved while using fewer edges and adding a small number of compressor nodes. The degree of nodes present in the original graph is also preserved. The current implementation of dedensification supports graphs with one edge type.

For more information on graph summarization, see Graph Summarization Methods and Applications: A Survey

`dedensify`(G, threshold[, prefix, copy])	Compresses neighborhoods around high-degree nodes
`snap_aggregation`(G, node_attributes[, ...])	Creates a summary graph based on attributes and connectivity.

3.59.1 dedensify

dedensify(*G*, *threshold*, *prefix=None*, *copy=True*)

Compresses neighborhoods around high-degree nodes

Reduces the number of edges to high-degree nodes by adding compressor nodes that summarize multiple edges of the same type to high-degree nodes (nodes with a degree greater than a given threshold). Dedensification also has the added benefit of reducing the number of edges around high-degree nodes. The implementation currently supports graphs with a single edge type.

Parameters

G: graph
A networkx graph

threshold: int
Minimum degree threshold of a node to be considered a high degree node. The threshold must be greater than or equal to 2.

prefix: str or None, optional (default: None)
An optional prefix for denoting compressor nodes

copy: bool, optional (default: True)
Indicates if dedensification should be done inplace

Returns

dedensified networkx graph
[(graph, set)] 2-tuple of the dedensified graph and set of compressor nodes

Notes

According to the algorithm in [1], removes edges in a graph by 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.

References

[1]

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"),
```

(continues on next page)

(continued from previous page)

```
...         ("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
```

3.59.2 snap_aggregation

snap_aggregation(*G*, *node_attributes*, *edge_attributes=()*, *prefix='Supernode-'*, *supernode_attribute='group'*, *superedge_attribute='types'*)

Creates a summary graph based on attributes and connectivity.

This function uses the Summarization by Grouping Nodes on Attributes and Pairwise edges (SNAP) algorithm for summarizing a given graph by grouping nodes by node attributes and their edge attributes into supernodes in a summary graph. This name SNAP should not be confused with the Stanford Network Analysis Project (SNAP).

Here is a high-level view of how this algorithm works:

1) Group nodes by node attribute values.

2) Iteratively split groups until all nodes in each group have edges to nodes in the same groups. That is, until all the groups are homogeneous in their member nodes' edges to other groups. For example, if all the nodes in group A only have edge to nodes in group B, then the group is homogeneous and does not need to be split. If all nodes in group B have edges with nodes in groups {A, C}, but some also have edges with other nodes in B, then group B is not homogeneous and needs to be split into groups have edges with {A, C} and a group of nodes having edges with {A, B, C}. This way, viewers of the summary graph can assume that all nodes in the group have the exact same node attributes and the exact same edges.

3) Build the output summary graph, where the groups are represented by super-nodes. Edges represent the edges shared between all the nodes in each respective groups.

A SNAP summary graph can be used to visualize graphs that are too large to display or visually analyze, or to efficiently identify sets of similar nodes with similar connectivity patterns to other sets of similar nodes based on specified node and/or edge attributes in a graph.

Parameters

G: graph
Networkx Graph to be summarized

edge_attributes: iterable, optional
An iterable of the edge attributes considered in the summarization process. If provided, unique combinations of the attribute values found in the graph are used to determine the edge types in the graph. If not provided, all edges are considered to be of the same type.

prefix: str
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

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]

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)
```

3.60 Swap

Swap edges in a graph.

double_edge_swap(G[, nswap, max_tries, seed])	Swap two edges in the graph while keeping the node degrees fixed.
directed_edge_swap(G, *[, nswap, max_tries, ...])	Swap three edges in a directed graph while keeping the node degrees fixed.
connected_double_edge_swap(G[, nswap, ...])	Attempts the specified number of double-edge swaps in the graph G.

3.60.1 double_edge_swap

double_edge_swap(*G*, *nswap=1*, *max_tries=100*, *seed=None*)

Swap two edges in the graph while keeping the node degrees fixed.

A double-edge swap removes two randomly chosen edges u-v and x-y and creates the new edges u-x and v-y:

```
u--v            u  v
       becomes  |  |
x--y            x  y
```

If either the edge u-x or v-y already exist no swap is performed and another attempt is made to find a suitable edge pair.

Parameters

G
[graph] An undirected graph

nswap
[integer (optional, default=1)] Number of double-edge swaps to perform

max_tries
[integer (optional)] Maximum number of attempts to swap edges

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

G
[graph] The graph after double edge swaps.

Raises

NetworkXError
If `G` is directed, or If `nswap` > `max_tries`, or If there are fewer than 4 nodes or 2 edges in `G`.

NetworkXAlgorithmError
If the number of swap attempts exceeds `max_tries` before `nswap` swaps are made

Notes

Does not enforce any connectivity constraints.

The graph G is modified in place.

3.60.2 directed_edge_swap

directed_edge_swap(*G*, ***, *nswap=1*, *max_tries=100*, *seed=None*)

Swap three edges in a directed graph while keeping the node degrees fixed.

A directed edge swap swaps three edges such that a -> b -> c -> d becomes a -> c -> b -> d. This pattern of swapping allows all possible states with the same in- and out-degree distribution in a directed graph to be reached.

If the swap would create parallel edges (e.g. if a -> c already existed in the previous example), another attempt is made to find a suitable trio of edges.

Parameters

G
[DiGraph] A directed graph

nswap
[integer (optional, default=1)] Number of three-edge (directed) swaps to perform

max_tries
[integer (optional, default=100)] Maximum number of attempts to swap edges

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

G
[DiGraph] The graph after the edges are swapped.

Raises

NetworkXError
If `G` is not directed, or If nswap > max_tries, or If there are fewer than 4 nodes or 3 edges in `G`.

NetworkXAlgorithmError
If the number of swap attempts exceeds `max_tries` before `nswap` swaps are made

Notes

Does not enforce any connectivity constraints.

The graph G is modified in place.

References

[1], [2]

3.60.3 connected_double_edge_swap

connected_double_edge_swap(*G*, *nswap=1*, *_window_threshold=3*, *seed=None*)

Attempts the specified number of double-edge swaps in the graph `G`.

A double-edge swap removes two randomly chosen edges `(u, v)` and `(x, y)` and creates the new edges `(u, x)` and `(v, y)`:

```
u--v            u  v
       becomes  |  |
x--y            x  y
```

If either `(u, x)` or `(v, y)` already exist, then no swap is performed so the actual number of swapped edges is always *at most* `nswap`.

Parameters

G
[graph] An undirected graph

nswap
[integer (optional, default=1)] Number of double-edge swaps to perform

_window_threshold
[integer] The window size below which connectedness of the graph will be checked after each swap.

The "window" in this function is a dynamically updated integer that represents the number of swap attempts to make before checking if the graph remains connected. It is an optimization used to decrease the running time of the algorithm in exchange for increased complexity of implementation.

If the window size is below this threshold, then the algorithm checks after each swap if the graph remains connected by checking if there is a path joining the two nodes whose edge was just removed. If the window size is above this threshold, then the algorithm performs do all the swaps in the window and only then check if the graph is still connected.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

int
The number of successful swaps

Raises

NetworkXError
If the input graph is not connected, or if the graph has fewer than four nodes.

Notes

The initial graph G must be connected, and the resulting graph is connected. The graph G is modified in place.

References

[1]

3.61 Threshold Graphs

Threshold Graphs - Creation, manipulation and identification.

find_threshold_graph(G[, create_using])	Returns a threshold subgraph that is close to largest in G.
is_threshold_graph(G)	Returns True if G is a threshold graph.

3.61.1 find_threshold_graph

find_threshold_graph(*G*, *create_using=None*)

Returns a threshold subgraph that is close to largest in G.

The threshold graph will contain the largest degree node in G.

Parameters

G
[NetworkX graph instance] An instance of Graph, or MultiDiGraph

create_using
[NetworkX graph class or None (default), optional] Type of graph to use when constructing the threshold graph. If None, infer the appropriate graph type from the input.

Returns

graph
A graph instance representing the threshold graph

References

[1]

Examples

```
>>> from networkx.algorithms.threshold import find_threshold_graph
>>> G = nx.barbell_graph(3, 3)
>>> T = find_threshold_graph(G)
>>> T.nodes # may vary
NodeView((7, 8, 5, 6))
```

3.61.2 is_threshold_graph

is_threshold_graph(*G*)

Returns `True` if `G` is a threshold graph.

Parameters

G

[NetworkX graph instance] An instance of `Graph`, `DiGraph`, `MultiGraph` or `MultiDiGraph`

Returns

bool

`True` if `G` is a threshold graph, `False` otherwise.

References

[1]

Examples

```
>>> from networkx.algorithms.threshold import is_threshold_graph
>>> G = nx.path_graph(3)
>>> is_threshold_graph(G)
True
>>> G = nx.barbell_graph(3, 3)
>>> is_threshold_graph(G)
False
```

3.62 Tournament

Functions concerning tournament graphs.

A tournament graph is a complete oriented graph. In other words, it is a directed graph in which there is exactly one directed edge joining each pair of distinct nodes. For each function in this module that accepts a graph as input, you must provide a tournament graph. The responsibility is on the caller to ensure that the graph is a tournament graph.

To access the functions in this module, you must access them through the *networkx.algorithms.tournament* module:

```
>>> from networkx.algorithms import tournament
>>> G = nx.DiGraph([(0, 1), (1, 2), (2, 0)])
>>> tournament.is_tournament(G)
True
```

hamiltonian_path(G)	Returns a Hamiltonian path in the given tournament graph.
is_reachable(G, s, t)	Decides whether there is a path from `s` to `t` in the tournament.
is_strongly_connected(G)	Decides whether the given tournament is strongly connected.
is_tournament(G)	Returns True if and only if `G` is a tournament.
random_tournament(n[, seed])	Returns a random tournament graph on `n` nodes.
score_sequence(G)	Returns the score sequence for the given tournament graph.

3.62.1 hamiltonian_path

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

path
[list] A list of nodes which form a Hamiltonian path in `G`.

Notes

This is a recursive implementation with an asymptotic running time of $O(n^2)$, ignoring multiplicative polylogarithmic factors, where n is the number of nodes in the graph.

Examples

```
>>> from networkx.algorithms import tournament
>>> G = nx.DiGraph([(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)])
>>> tournament.hamiltonian_path(G)
[0, 1, 2, 3]
```

3.62.2 is_reachable

is_reachable(*G*, *s*, *t*)

Decides whether there is a path from `s` to `t` in the tournament.

This function is more theoretically efficient than the reachability checks than the shortest path algorithms in `networkx.algorithms.shortest_paths`.

The given graph **must** be a tournament, otherwise this function's behavior is undefined.

Parameters

G
[NetworkX graph] A directed graph representing a tournament.

s
[node] A node in the graph.

t
[node] A node in the graph.

Returns

bool
Whether there is a path from `s` to `t` in `G`.

Notes

Although this function is more theoretically efficient than the generic shortest path functions, a speedup requires the use of parallelism. Though it may in the future, the current implementation does not use parallelism, thus you may not see much of a speedup.

This algorithm comes from [1].

References

[1]

Examples

```
>>> from networkx.algorithms import tournament
>>> G = nx.DiGraph([(1, 0), (1, 3), (1, 2), (2, 3), (2, 0), (3, 0)])
>>> tournament.is_reachable(G, 1, 3)
True
>>> tournament.is_reachable(G, 3, 2)
False
```

3.62.3 is_strongly_connected

is_strongly_connected(*G*)

Decides whether the given tournament is strongly connected.

This function is more theoretically efficient than the *is_strongly_connected()* function.

The given graph **must** be a tournament, otherwise this function's behavior is undefined.

Parameters

G
[NetworkX graph] A directed graph representing a tournament.

Returns

bool
Whether the tournament is strongly connected.

Notes

Although this function is more theoretically efficient than the generic strong connectivity function, a speedup requires the use of parallelism. Though it may in the future, the current implementation does not use parallelism, thus you may not see much of a speedup.

This algorithm comes from [1].

References

[1]

Examples

```
>>> from networkx.algorithms import tournament
>>> G = nx.DiGraph([(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3), (3, 0)])
>>> tournament.is_strongly_connected(G)
True
>>> G.remove_edge(1, 3)
>>> tournament.is_strongly_connected(G)
False
```

3.62.4 is_tournament

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.

Notes

Some definitions require a self-loop on each node, but that is not the convention used here.

Examples

```
>>> from networkx.algorithms import tournament
>>> G = nx.DiGraph([(0, 1), (1, 2), (2, 0)])
>>> tournament.is_tournament(G)
True
```

3.62.5 random_tournament

random_tournament(*n*, *seed=None*)

Returns a random tournament graph on `n` nodes.

Parameters

n
[int] The number of nodes in the returned graph.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

G
[DiGraph] A tournament on `n` nodes, with exactly one directed edge joining each pair of distinct nodes.

Notes

This algorithm adds, for each pair of distinct nodes, an edge with uniformly random orientation. In other words, `binom{n}{2}` flips of an unbiased coin decide the orientations of the edges in the graph.

3.62.6 score_sequence

score_sequence(*G*)

Returns the score sequence for the given tournament graph.

The score sequence is the sorted list of the out-degrees of the nodes of the graph.

Parameters

G
[NetworkX graph] A directed graph representing a tournament.

Returns

list
A sorted list of the out-degrees of the nodes of `G`.

Examples

```
>>> from networkx.algorithms import tournament
>>> G = nx.DiGraph([(1, 0), (1, 3), (0, 2), (0, 3), (2, 1), (3, 2)])
>>> tournament.score_sequence(G)
[1, 1, 2, 2]
```

3.63 Traversal

3.63.1 Depth First Search

Basic algorithms for depth-first searching the nodes of a graph.

dfs_edges(G[, source, depth_limit])	Iterate over edges in a depth-first-search (DFS).
dfs_tree(G[, source, depth_limit])	Returns oriented tree constructed from a depth-first-search from source.
dfs_predecessors(G[, source, depth_limit])	Returns dictionary of predecessors in depth-first-search from source.
dfs_successors(G[, source, depth_limit])	Returns dictionary of successors in depth-first-search from source.
dfs_preorder_nodes(G[, source, depth_limit])	Generate nodes in a depth-first-search pre-ordering starting at source.
dfs_postorder_nodes(G[, source, depth_limit])	Generate nodes in a depth-first-search post-ordering starting at source.
dfs_labeled_edges(G[, source, depth_limit])	Iterate over edges in a depth-first-search (DFS) labeled by type.

dfs_edges

dfs_edges(*G*, *source=None*, *depth_limit=None*)

Iterate over edges in a depth-first-search (DFS).

Perform a depth-first-search over the nodes of `G` and yield the edges in order. This may not generate all edges in `G` (see *edge_dfs*).

Parameters

G
[NetworkX graph]

source
[node, optional] Specify starting node for depth-first search and yield edges in the component reachable from source.

depth_limit
[int, optional (default=len(G))] Specify the maximum search depth.

Yields

edge: 2-tuple of nodes
Yields edges resulting from the depth-first-search.

See also:

dfs_preorder_nodes
dfs_postorder_nodes
dfs_labeled_edges
edge_dfs()
bfs_edges()

Notes

If a source is not specified then a source is chosen arbitrarily and repeatedly until all components in the graph are searched.

The implementation of this function is adapted from David Eppstein's depth-first search function in PADS [1], with modifications to allow depth limits based on the Wikipedia article "Depth-limited search" [2].

References

[1], [2]

Examples

```
>>> G = nx.path_graph(5)
>>> list(nx.dfs_edges(G, source=0))
[(0, 1), (1, 2), (2, 3), (3, 4)]
>>> list(nx.dfs_edges(G, source=0, depth_limit=2))
[(0, 1), (1, 2)]
```

dfs_tree

dfs_tree(*G*, *source=None*, *depth_limit=None*)

Returns oriented tree constructed from a depth-first-search from source.

Parameters

G
[NetworkX graph]

source
[node, optional] Specify starting node for depth-first search.

depth_limit
[int, optional (default=len(G))] Specify the maximum search depth.

Returns

T
[NetworkX DiGraph] An oriented tree

See also:

dfs_preorder_nodes
dfs_postorder_nodes
dfs_labeled_edges
edge_dfs
bfs_tree

Examples

```
>>> G = nx.path_graph(5)
>>> T = nx.dfs_tree(G, source=0, depth_limit=2)
>>> list(T.edges())
[(0, 1), (1, 2)]
>>> T = nx.dfs_tree(G, source=0)
>>> list(T.edges())
[(0, 1), (1, 2), (2, 3), (3, 4)]
```

dfs_predecessors

dfs_predecessors(*G*, *source=None*, *depth_limit=None*)

Returns dictionary of predecessors in depth-first-search from source.

Parameters

G
[NetworkX graph]

source
[node, optional] Specify starting node for depth-first search.

depth_limit
[int, optional (default=len(G))] Specify the maximum search depth.

Returns

pred: dict
A dictionary with nodes as keys and predecessor nodes as values.

See also:

dfs_preorder_nodes
dfs_postorder_nodes
dfs_labeled_edges
edge_dfs
bfs_tree

Notes

If a source is not specified then a source is chosen arbitrarily and repeatedly until all components in the graph are searched.

The implementation of this function is adapted from David Eppstein's depth-first search function in PADS, with modifications to allow depth limits based on the Wikipedia article "Depth-limited search".

Examples

```
>>> G = nx.path_graph(4)
>>> nx.dfs_predecessors(G, source=0)
{1: 0, 2: 1, 3: 2}
>>> nx.dfs_predecessors(G, source=0, depth_limit=2)
{1: 0, 2: 1}
```

dfs_successors

dfs_successors(*G*, *source=None*, *depth_limit=None*)

Returns dictionary of successors in depth-first-search from source.

Parameters

G
[NetworkX graph]

source
[node, optional] Specify starting node for depth-first search.

depth_limit
[int, optional (default=len(G))] Specify the maximum search depth.

Returns

succ: dict
A dictionary with nodes as keys and list of successor nodes as values.

See also:

dfs_preorder_nodes
dfs_postorder_nodes
dfs_labeled_edges
edge_dfs
bfs_tree

Notes

If a source is not specified then a source is chosen arbitrarily and repeatedly until all components in the graph are searched.

The implementation of this function is adapted from David Eppstein's depth-first search function in PADS, with modifications to allow depth limits based on the Wikipedia article "Depth-limited search".

Examples

```
>>> G = nx.path_graph(5)
>>> nx.dfs_successors(G, source=0)
{0: [1], 1: [2], 2: [3], 3: [4]}
>>> nx.dfs_successors(G, source=0, depth_limit=2)
{0: [1], 1: [2]}
```

dfs_preorder_nodes

dfs_preorder_nodes(*G*, *source=None*, *depth_limit=None*)

Generate nodes in a depth-first-search pre-ordering starting at source.

Parameters

G
[NetworkX graph]

source
[node, optional] Specify starting node for depth-first search and return nodes in the component reachable from source.

depth_limit
[int, optional (default=len(G))] Specify the maximum search depth.

Returns

nodes: generator
A generator of nodes in a depth-first-search pre-ordering.

See also:

dfs_edges
dfs_postorder_nodes
dfs_labeled_edges
bfs_edges

Notes

If a source is not specified then a source is chosen arbitrarily and repeatedly until all components in the graph are searched.

The implementation of this function is adapted from David Eppstein's depth-first search function in PADS, with modifications to allow depth limits based on the Wikipedia article "Depth-limited search".

Examples

```
>>> G = nx.path_graph(5)
>>> list(nx.dfs_preorder_nodes(G, source=0))
[0, 1, 2, 3, 4]
>>> list(nx.dfs_preorder_nodes(G, source=0, depth_limit=2))
[0, 1, 2]
```

dfs_postorder_nodes

dfs_postorder_nodes(*G*, *source=None*, *depth_limit=None*)

Generate nodes in a depth-first-search post-ordering starting at source.

Parameters

G
[NetworkX graph]

source
[node, optional] Specify starting node for depth-first search.

depth_limit
[int, optional (default=len(G))] Specify the maximum search depth.

Returns

nodes: generator
A generator of nodes in a depth-first-search post-ordering.

See also:

dfs_edges
dfs_preorder_nodes
dfs_labeled_edges
edge_dfs
bfs_tree

Notes

If a source is not specified then a source is chosen arbitrarily and repeatedly until all components in the graph are searched.

The implementation of this function is adapted from David Eppstein's depth-first search function in PADS, with modifications to allow depth limits based on the Wikipedia article "Depth-limited search".

Examples

```
>>> G = nx.path_graph(5)
>>> list(nx.dfs_postorder_nodes(G, source=0))
[4, 3, 2, 1, 0]
>>> list(nx.dfs_postorder_nodes(G, source=0, depth_limit=2))
[1, 0]
```

dfs_labeled_edges

dfs_labeled_edges(*G*, *source=None*, *depth_limit=None*)

Iterate over edges in a depth-first-search (DFS) labeled by type.

Parameters

G
[NetworkX graph]

source
[node, optional] Specify starting node for depth-first search and return edges in the component reachable from source.

depth_limit
[int, optional (default=len(G))] Specify the maximum search depth.

Returns

edges: generator
A generator of triples of the form (u, v, d), where (u, v) is the edge being explored in the depth-first search and d is one of the strings 'forward', 'nontree', 'reverse', or 'reverse-depth_limit'. A 'forward' edge is one in which u has been visited but v has not. A 'nontree' edge is one in which both u and v have been visited but the edge is not in the DFS tree. A 'reverse' edge is one in which both u and v have been visited and the edge is in the DFS tree. When the `depth_limit` is reached via a 'forward' edge, a 'reverse' edge is immediately generated rather than the subtree being explored. To indicate this flavor of 'reverse' edge, the string yielded is 'reverse-depth_limit'.

See also:

dfs_edges
dfs_preorder_nodes
dfs_postorder_nodes

Notes

If a source is not specified then a source is chosen arbitrarily and repeatedly until all components in the graph are searched.

The implementation of this function is adapted from David Eppstein's depth-first search function in PADS, with modifications to allow depth limits based on the Wikipedia article "Depth-limited search".

Examples

The labels reveal the complete transcript of the depth-first search algorithm in more detail than, for example, *dfs_edges()*:

```
>>> from pprint import pprint
>>>
>>> G = nx.DiGraph([(0, 1), (1, 2), (2, 1)])
>>> pprint(list(nx.dfs_labeled_edges(G, source=0)))
[(0, 0, 'forward'),
 (0, 1, 'forward'),
 (1, 2, 'forward'),
 (2, 1, 'nontree'),
 (1, 2, 'reverse'),
 (0, 1, 'reverse'),
 (0, 0, 'reverse')]
```

3.63.2 Breadth First Search

Basic algorithms for breadth-first searching the nodes of a graph.

bfs_edges(G, source[, reverse, depth_limit, ...])	Iterate over edges in a breadth-first-search starting at source.
bfs_layers(G, sources)	Returns an iterator of all the layers in breadth-first search traversal.
bfs_tree(G, source[, reverse, depth_limit, ...])	Returns an oriented tree constructed from of a breadth-first-search starting at source.
bfs_predecessors(G, source[, depth_limit, ...])	Returns an iterator of predecessors in breadth-first-search from source.
bfs_successors(G, source[, depth_limit, ...])	Returns an iterator of successors in breadth-first-search from source.
descendants_at_distance(G, source, distance)	Returns all nodes at a fixed `distance` from `source` in `G`.

bfs_edges

bfs_edges(*G*, *source*, *reverse=False*, *depth_limit=None*, *sort_neighbors=None*)

Iterate over edges in a breadth-first-search starting at source.

Parameters

G
[NetworkX graph]

source
[node] Specify starting node for breadth-first search; this function iterates over only those edges in the component reachable from this node.

reverse
[bool, optional] If True traverse a directed graph in the reverse direction

depth_limit
[int, optional(default=len(G))] Specify the maximum search depth

sort_neighbors
[function] A function that takes the list of neighbors of given node as input, and returns an *iterator* over these neighbors but with custom ordering.

Yields

edge: 2-tuple of nodes
Yields edges resulting from the breadth-first search.

See also:

bfs_tree
dfs_edges()
edge_bfs()

Notes

The naming of this function is very similar to *edge_bfs()*. The difference is that edge_bfs yields edges even if they extend back to an already explored node while this generator yields the edges of the tree that results from a breadth-first-search (BFS) so no edges are reported if they extend to already explored nodes. That means edge_bfs reports all edges while bfs_edges only reports those traversed by a node-based BFS. Yet another description is that bfs_edges reports the edges traversed during BFS while edge_bfs reports all edges in the order they are explored.

Based on the breadth-first search implementation in PADS [1] by D. Eppstein, July 2004; with modifications to allow depth limits as described in [2].

References

[1], [2]

Examples

To get the edges in a breadth-first search:

```
>>> G = nx.path_graph(3)
>>> list(nx.bfs_edges(G, 0))
[(0, 1), (1, 2)]
>>> list(nx.bfs_edges(G, source=0, depth_limit=1))
[(0, 1)]
```

To get the nodes in a breadth-first search order:

```
>>> G = nx.path_graph(3)
>>> root = 2
>>> edges = nx.bfs_edges(G, root)
>>> nodes = [root] + [v for u, v in edges]
>>> nodes
[2, 1, 0]
```

bfs_layers

bfs_layers(*G*, *sources*)

Returns an iterator of all the layers in breadth-first search traversal.

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] Specify starting nodes for single source or multiple sources breadth-first search

Yields

layer: list of nodes
Yields list of nodes at the same distance from sources

Examples

```
>>> G = nx.path_graph(5)
>>> dict(enumerate(nx.bfs_layers(G, [0, 4])))
{0: [0, 4], 1: [1, 3], 2: [2]}
>>> H = nx.Graph()
>>> H.add_edges_from([(0, 1), (0, 2), (1, 3), (1, 4), (2, 5), (2, 6)])
>>> dict(enumerate(nx.bfs_layers(H, [1])))
{0: [1], 1: [0, 3, 4], 2: [2], 3: [5, 6]}
>>> dict(enumerate(nx.bfs_layers(H, [1, 6])))
{0: [1, 6], 1: [0, 3, 4, 2], 2: [5]}
```

bfs_tree

bfs_tree(*G*, *source*, *reverse=False*, *depth_limit=None*, *sort_neighbors=None*)

Returns an oriented tree constructed from of a breadth-first-search starting at source.

Parameters

G
[NetworkX graph]

source
[node] Specify starting node for breadth-first search

reverse
[bool, optional] If True traverse a directed graph in the reverse direction

depth_limit
[int, optional(default=len(G))] Specify the maximum search depth

sort_neighbors
[function] A function that takes the list of neighbors of given node as input, and returns an *iterator* over these neighbors but with custom ordering.

Returns

T: NetworkX DiGraph
An oriented tree

See also:

dfs_tree
bfs_edges
edge_bfs

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".

Examples

```
>>> G = nx.path_graph(3)
>>> print(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])
>>> print(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)]
```

bfs_predecessors

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] A function that takes the list of neighbors of given node as input, and returns an *iterator* over these neighbors but with custom ordering.

Returns

pred: iterator
(node, predecessor) iterator where `predecessor` is the predecessor of `node` in a breadth first search starting from `source`.

See also:

bfs_tree
bfs_edges
edge_bfs

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".

Examples

```
>>> G = nx.path_graph(3)
>>> print(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)])
>>> print(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])
>>> print(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()
>>> nx.add_path(N, [0, 1, 2, 3, 4, 7])
>>> nx.add_path(N, [3, 5, 6, 7])
>>> print(sorted(nx.bfs_predecessors(N, source=2)))
[(3, 2), (4, 3), (5, 3), (6, 5), (7, 4)]
```

bfs_successors

bfs_successors(*G*, *source*, *depth_limit=None*, *sort_neighbors=None*)

Returns an iterator of successors 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] A function that takes the list of neighbors of given node as input, and returns an *iterator* over these neighbors but with custom ordering.

Returns

succ: iterator
(node, successors) iterator where `successors` is the non-empty list of successors of `node` in a breadth first search from `source`. To appear in the iterator, `node` must have successors.

See also:

bfs_tree
bfs_edges
edge_bfs

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".

Examples

```
>>> G = nx.path_graph(3)
>>> print(dict(nx.bfs_successors(G, 0)))
{0: [1], 1: [2]}
>>> H = nx.Graph()
>>> H.add_edges_from([(0, 1), (0, 2), (1, 3), (1, 4), (2, 5), (2, 6)])
>>> print(dict(nx.bfs_successors(H, 0)))
{0: [1, 2], 1: [3, 4], 2: [5, 6]}
>>> G = nx.Graph()
>>> nx.add_path(G, [0, 1, 2, 3, 4, 5, 6])
>>> nx.add_path(G, [2, 7, 8, 9, 10])
>>> print(dict(nx.bfs_successors(G, source=1, depth_limit=3)))
{1: [0, 2], 2: [3, 7], 3: [4], 7: [8]}
>>> G = nx.DiGraph()
>>> nx.add_path(G, [0, 1, 2, 3, 4, 5])
>>> print(dict(nx.bfs_successors(G, source=3)))
{3: [4], 4: [5]}
```

descendants_at_distance

descendants_at_distance(*G*, *source*, *distance*)

Returns all nodes at a fixed `distance` from `source` in `G`.

Parameters

G
[NetworkX graph] A graph

source
[node in `G`]

distance
[the distance of the wanted nodes from `source`]

Returns

set()
The descendants of `source` in `G` at the given `distance` from `source`

Examples

```
>>> G = nx.path_graph(5)
>>> nx.descendants_at_distance(G, 2, 2)
{0, 4}
>>> H = nx.DiGraph()
>>> H.add_edges_from([(0, 1), (0, 2), (1, 3), (1, 4), (2, 5), (2, 6)])
>>> nx.descendants_at_distance(H, 0, 2)
{3, 4, 5, 6}
```

(continues on next page)

(continued from previous page)

```
>>> nx.descendants_at_distance(H, 5, 0)
{5}
>>> nx.descendants_at_distance(H, 5, 1)
set()
```

3.63.3 Beam search

Basic algorithms for breadth-first searching the nodes of a graph.

bfs_beam_edges(G, source, value[, width])	Iterates over edges in a beam search.

bfs_beam_edges

bfs_beam_edges(*G*, *source*, *value*, *width=None*)

Iterates over edges in a beam search.

The beam search is a generalized breadth-first search in which only the "best" *w* neighbors of the current node are enqueued, where *w* is the beam width and "best" is an application-specific heuristic. In general, a beam search with a small beam width might not visit each node in the graph.

Parameters

G
[NetworkX graph]

source
[node] Starting node for the breadth-first search; this function iterates over only those edges in the component reachable from this node.

value
[function] A function that takes a node of the graph as input and returns a real number indicating how "good" it is. A higher value means it is more likely to be visited sooner during the search. When visiting a new node, only the `width` neighbors with the highest `value` are enqueued (in decreasing order of `value`).

width
[int (default = None)] The beam width for the search. This is the number of neighbors (ordered by `value`) to enqueue when visiting each new node.

Yields

edge
Edges in the beam search starting from `source`, given as a pair of nodes.

Examples

To give nodes with, for example, a higher centrality precedence during the search, set the `value` function to return the centrality value of the node:

```
>>> G = nx.karate_club_graph()
>>> centrality = nx.eigenvector_centrality(G)
>>> source = 0
>>> width = 5
>>> for u, v in nx.bfs_beam_edges(G, source, centrality.get, width):
...     print((u, v))
...
(0, 2)
(0, 1)
(0, 8)
(0, 13)
(0, 3)
(2, 32)
(1, 30)
(8, 33)
(3, 7)
(32, 31)
(31, 28)
(31, 25)
(25, 23)
(25, 24)
(23, 29)
(23, 27)
(29, 26)
```

3.63.4 Depth First Search on Edges

Algorithms for a depth-first traversal of edges in a graph.

edge_dfs(G[, source, orientation])	A directed, depth-first-search of edges in `G`, beginning at `source`.

edge_dfs

edge_dfs(*G*, *source=None*, *orientation=None*)

A directed, depth-first-search of edges in `G`, beginning at `source`.

Yield the edges of G in a depth-first-search order continuing until all edges are generated.

Parameters

G
[graph] A directed/undirected graph/multigraph.

source
[node, list of nodes] The node from which the traversal begins. If None, then a source is chosen arbitrarily and repeatedly until all edges from each node in the graph are searched.

orientation
[None | 'original' | 'reverse' | 'ignore' (default: None)] For directed graphs and directed multigraphs, edge traversals need not respect the original orientation of the edges. When set to

'reverse' every edge is traversed in the reverse direction. When set to 'ignore', every edge is treated as undirected. When set to 'original', every edge is treated as directed. In all three cases, the yielded edge tuples add a last entry to indicate the direction in which that edge was traversed. If orientation is None, the yielded edge has no direction indicated. The direction is respected, but not reported.

Yields

edge

[directed edge] A directed edge indicating the path taken by the depth-first traversal. For graphs, `edge` is of the form `(u, v)` where `u` and `v` are the tail and head of the edge as determined by the traversal. For multigraphs, `edge` is of the form `(u, v, key)`, where `key` is the key of the edge. When the graph is directed, then `u` and `v` are always in the order of the actual directed edge. If orientation is not None then the edge tuple is extended to include the direction of traversal ('forward' or 'reverse') on that edge.

See also:

dfs_edges()

Notes

The goal of this function is to visit edges. It differs from the more familiar depth-first traversal of nodes, as provided by *dfs_edges()*, in that it does not stop once every node has been visited. In a directed graph with edges [(0, 1), (1, 2), (2, 1)], the edge (2, 1) would not be visited if not for the functionality provided by this function.

Examples

```
>>> nodes = [0, 1, 2, 3]
>>> edges = [(0, 1), (1, 0), (1, 0), (2, 1), (3, 1)]
```

```
>>> list(nx.edge_dfs(nx.Graph(edges), nodes))
[(0, 1), (1, 2), (1, 3)]
```

```
>>> list(nx.edge_dfs(nx.DiGraph(edges), nodes))
[(0, 1), (1, 0), (2, 1), (3, 1)]
```

```
>>> list(nx.edge_dfs(nx.MultiGraph(edges), nodes))
[(0, 1, 0), (1, 0, 1), (0, 1, 2), (1, 2, 0), (1, 3, 0)]
```

```
>>> list(nx.edge_dfs(nx.MultiDiGraph(edges), nodes))
[(0, 1, 0), (1, 0, 0), (1, 0, 1), (2, 1, 0), (3, 1, 0)]
```

```
>>> list(nx.edge_dfs(nx.DiGraph(edges), nodes, orientation="ignore"))
[(0, 1, 'forward'), (1, 0, 'forward'), (2, 1, 'reverse'), (3, 1, 'reverse')]
```

```
>>> list(nx.edge_dfs(nx.MultiDiGraph(edges), nodes, orientation="ignore"))
[(0, 1, 0, 'forward'), (1, 0, 0, 'forward'), (1, 0, 1, 'reverse'), (2, 1, 0,
↪'reverse'), (3, 1, 0, 'reverse')]
```

3.63.5 Breadth First Search on Edges

Algorithms for a breadth-first traversal of edges in a graph.

edge_bfs(G[, source, orientation])	A directed, breadth-first-search of edges in `G`, beginning at `source`.

edge_bfs

edge_bfs(*G*, *source=None*, *orientation=None*)

A directed, breadth-first-search of edges in `G`, beginning at `source`.

Yield the edges of G in a breadth-first-search order continuing until all edges are generated.

Parameters

G
[graph] A directed/undirected graph/multigraph.

source
[node, list of nodes] The node from which the traversal begins. If None, then a source is chosen arbitrarily and repeatedly until all edges from each node in the graph are searched.

orientation
[None | 'original' | 'reverse' | 'ignore' (default: None)] For directed graphs and directed multigraphs, edge traversals need not respect the original orientation of the edges. When set to 'reverse' every edge is traversed in the reverse direction. When set to 'ignore', every edge is treated as undirected. When set to 'original', every edge is treated as directed. In all three cases, the yielded edge tuples add a last entry to indicate the direction in which that edge was traversed. If orientation is None, the yielded edge has no direction indicated. The direction is respected, but not reported.

Yields

edge
[directed edge] A directed edge indicating the path taken by the breadth-first-search. For graphs, `edge` is of the form `(u, v)` where `u` and `v` are the tail and head of the edge as determined by the traversal. For multigraphs, `edge` is of the form `(u, v, key)`, where `key` is the key of the edge. When the graph is directed, then `u` and `v` are always in the order of the actual directed edge. If orientation is not None then the edge tuple is extended to include the direction of traversal ('forward' or 'reverse') on that edge.

See also:

bfs_edges
bfs_tree
edge_dfs

Notes

The goal of this function is to visit edges. It differs from the more familiar breadth-first-search of nodes, as provided by *networkx.algorithms.traversal.breadth_first_search.bfs_edges()*, in that it does not stop once every node has been visited. In a directed graph with edges [(0, 1), (1, 2), (2, 1)], the edge (2, 1) would not be visited if not for the functionality provided by this function.

The naming of this function is very similar to bfs_edges. The difference is that 'edge_bfs' yields edges even if they extend back to an already explored node while 'bfs_edges' yields the edges of the tree that results from a breadth-first-search (BFS) so no edges are reported if they extend to already explored nodes. That means 'edge_bfs' reports all edges while 'bfs_edges' only report those traversed by a node-based BFS. Yet another description is that 'bfs_edges' reports the edges traversed during BFS while 'edge_bfs' reports all edges in the order they are explored.

Examples

```
>>> nodes = [0, 1, 2, 3]
>>> edges = [(0, 1), (1, 0), (1, 0), (2, 0), (2, 1), (3, 1)]
```

```
>>> list(nx.edge_bfs(nx.Graph(edges), nodes))
[(0, 1), (0, 2), (1, 2), (1, 3)]
```

```
>>> list(nx.edge_bfs(nx.DiGraph(edges), nodes))
[(0, 1), (1, 0), (2, 0), (2, 1), (3, 1)]
```

```
>>> list(nx.edge_bfs(nx.MultiGraph(edges), nodes))
[(0, 1, 0), (0, 1, 1), (0, 1, 2), (0, 2, 0), (1, 2, 0), (1, 3, 0)]
```

```
>>> list(nx.edge_bfs(nx.MultiDiGraph(edges), nodes))
[(0, 1, 0), (1, 0, 0), (1, 0, 1), (2, 0, 0), (2, 1, 0), (3, 1, 0)]
```

```
>>> list(nx.edge_bfs(nx.DiGraph(edges), nodes, orientation="ignore"))
[(0, 1, 'forward'), (1, 0, 'reverse'), (2, 0, 'reverse'), (2, 1, 'reverse'), (3,␣
↪1, 'reverse')]
```

```
>>> list(nx.edge_bfs(nx.MultiDiGraph(edges), nodes, orientation="ignore"))
[(0, 1, 0, 'forward'), (1, 0, 0, 'reverse'), (1, 0, 1, 'reverse'), (2, 0, 0,
↪'reverse'), (2, 1, 0, 'reverse'), (3, 1, 0, 'reverse')]
```

3.64 Tree

3.64.1 Recognition

Recognition Tests

A *forest* is an acyclic, undirected graph, and a *tree* is a connected forest. Depending on the subfield, there are various conventions for generalizing these definitions to directed graphs.

In one convention, directed variants of forest and tree are defined in an identical manner, except that the direction of the edges is ignored. In effect, each directed edge is treated as a single undirected edge. Then, additional restrictions are imposed to define *branchings* and *arborescences*.

In another convention, directed variants of forest and tree correspond to the previous convention's branchings and arborescences, respectively. Then two new terms, *polyforest* and *polytree*, are defined to correspond to the other convention's forest and tree.

Summarizing:

Convention A	Convention B
forest	polyforest
tree	polytree
branching	forest
arborescence	tree

Each convention has its reasons. The first convention emphasizes definitional similarity in that directed forests and trees are only concerned with acyclicity and do not have an in-degree constraint, just as their undirected counterparts do not. The second convention emphasizes functional similarity in the sense that the directed analog of a spanning tree is a spanning arborescence. That is, take any spanning tree and choose one node as the root. Then every edge is assigned a direction such there is a directed path from the root to every other node. The result is a spanning arborescence.

NetworkX follows convention "A". Explicitly, these are:

undirected forest
: An undirected graph with no undirected cycles.

undirected tree
: A connected, undirected forest.

directed forest
: A directed graph with no undirected cycles. Equivalently, the underlying graph structure (which ignores edge orientations) is an undirected forest. In convention B, this is known as a polyforest.

directed tree
: A weakly connected, directed forest. Equivalently, the underlying graph structure (which ignores edge orientations) is an undirected tree. In convention B, this is known as a polytree.

branching
: A directed forest with each node having, at most, one parent. So the maximum in-degree is equal to 1. In convention B, this is known as a forest.

arborescence
: A directed tree with each node having, at most, one parent. So the maximum in-degree is equal to 1. In convention B, this is known as a tree.

For trees and arborescences, the adjective "spanning" may be added to designate that the graph, when considered as a forest/branching, consists of a single tree/arborescence that includes all nodes in the graph. It is true, by definition, that every tree/arborescence is spanning with respect to the nodes that define the tree/arborescence and so, it might seem redundant to introduce the notion of "spanning". However, the nodes may represent a subset of nodes from a larger graph, and it is in this context that the term "spanning" becomes a useful notion.

is_tree(G)	Returns True if G is a tree.
is_forest(G)	Returns True if G is a forest.
is_arborescence(G)	Returns True if G is an arborescence.
is_branching(G)	Returns True if G is a branching.

is_tree

is_tree(*G*)

Returns True if G is a tree.

A tree is a connected graph with no undirected cycles.

For directed graphs, G is a tree if the underlying graph is a tree. The underlying graph is obtained by treating each directed edge as a single undirected edge in a multigraph.

Parameters

G
[graph] The graph to test.

Returns

b
[bool] A boolean that is True if G is a tree.

Raises

NetworkXPointlessConcept
If G is empty.

See also:

is_arborescence

Notes

In another convention, a directed tree is known as a *polytree* and then *tree* corresponds to an *arborescence*.

Examples

```
>>> G = nx.Graph()
>>> G.add_edges_from([(1, 2), (1, 3), (2, 4), (2, 5)])
>>> nx.is_tree(G)  # n-1 edges
True
>>> G.add_edge(3, 4)
>>> nx.is_tree(G)  # n edges
False
```

is_forest

is_forest(*G*)

Returns True if G is a forest.

A forest is a graph with no undirected cycles.

For directed graphs, G is a forest if the underlying graph is a forest. The underlying graph is obtained by treating each directed edge as a single undirected edge in a multigraph.

Parameters

G
[graph] The graph to test.

Returns

b
[bool] A boolean that is True if G is a forest.

Raises

NetworkXPointlessConcept
If G is empty.

See also:

is_branching

Notes

In another convention, a directed forest is known as a *polyforest* and then *forest* corresponds to a *branching*.

Examples

```
>>> G = nx.Graph()
>>> G.add_edges_from([(1, 2), (1, 3), (2, 4), (2, 5)])
>>> nx.is_forest(G)
True
>>> G.add_edge(4, 1)
>>> nx.is_forest(G)
False
```

is_arborescence

is_arborescence(*G*)

Returns True if G is an arborescence.

An arborescence is a directed tree with maximum in-degree equal to 1.

Parameters

G
[graph] The graph to test.

Returns

b
[bool] A boolean that is True if G is an arborescence.

See also:

is_tree

Notes

In another convention, an arborescence is known as a *tree*.

Examples

```
>>> G = nx.DiGraph([(0, 1), (0, 2), (2, 3), (3, 4)])
>>> nx.is_arborescence(G)
True
>>> G.remove_edge(0, 1)
>>> G.add_edge(1, 2)  # maximum in-degree is 2
>>> nx.is_arborescence(G)
False
```

is_branching

is_branching(*G*)

Returns True if `G` is a branching.

A branching is a directed forest with maximum in-degree equal to 1.

Parameters

G
[directed graph] The directed graph to test.

Returns

b
[bool] A boolean that is True if `G` is a branching.

See also:

is_forest

Notes

In another convention, a branching is also known as a *forest*.

Examples

```
>>> G = nx.DiGraph([(0, 1), (1, 2), (2, 3), (3, 4)])
>>> nx.is_branching(G)
True
>>> G.remove_edge(2, 3)
>>> G.add_edge(3, 1)  # maximum in-degree is 2
>>> nx.is_branching(G)
False
```

3.64.2 Branchings and Spanning Arborescences

Algorithms for finding optimum branchings and spanning arborescences.

This implementation is based on:

> J. Edmonds, Optimum branchings, J. Res. Natl. Bur. Standards 71B (1967), 233–240. URL: http://archive.org/details/jresv71Bn4p233

branching_weight(G[, attr, default])	Returns the total weight of a branching.
greedy_branching(G[, attr, default, kind, seed])	Returns a branching obtained through a greedy algorithm.
maximum_branching(G[, attr, default, ...])	Returns a maximum branching from G.
minimum_branching(G[, attr, default, ...])	Returns a minimum branching from G.
maximum_spanning_arborescence(G[, attr, ...])	Returns a maximum spanning arborescence from G.
minimum_spanning_arborescence(G[, attr, ...])	Returns a minimum spanning arborescence from G.
ArborescenceIterator(G[, weight, minimum, ...])	Iterate over all spanning arborescences of a graph in either increasing or decreasing cost.
Edmonds(G[, seed])	Edmonds algorithm [1] for finding optimal branchings and spanning arborescences.

branching_weight

branching_weight(*G*, *attr='weight'*, *default=1*)

Returns the total weight of a branching.

You must access this function through the networkx.algorithms.tree module.

Parameters

G
[DiGraph] The directed graph.

attr
[str] The attribute to use as weights. If None, then each edge will be treated equally with a weight of 1.

default
[float] When `attr` is not None, then if an edge does not have that attribute, `default` specifies what value it should take.

Returns

weight: int or float
The total weight of the branching.

Examples

```
>>> G = nx.DiGraph()
>>> G.add_weighted_edges_from([(0, 1, 2), (1, 2, 4), (2, 3, 3), (3, 4, 2)])
>>> nx.tree.branching_weight(G)
11
```

greedy_branching

greedy_branching(*G*, *attr='weight'*, *default=1*, *kind='max'*, *seed=None*)

Returns a branching obtained through a greedy algorithm.

This algorithm is wrong, and cannot give a proper optimal branching. However, we include it for pedagogical reasons, as it can be helpful to see what its outputs are.

The output is a branching, and possibly, a spanning arborescence. However, it is not guaranteed to be optimal in either case.

Parameters

G
[DiGraph] The directed graph to scan.

attr
[str] The attribute to use as weights. If None, then each edge will be treated equally with a weight of 1.

default
[float] When `attr` is not None, then if an edge does not have that attribute, `default` specifies what value it should take.

kind
[str] The type of optimum to search for: 'min' or 'max' greedy branching.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

B
[directed graph] The greedily obtained branching.

maximum_branching

maximum_branching(*G*, *attr='weight'*, *default=1*, *preserve_attrs=False*, *partition=None*)

Returns a maximum branching from G.

Parameters

G
[(multi)digraph-like] The graph to be searched.

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`.

preserve_attrs
[bool] If True, preserve the other attributes of the original graph (that are not passed to `attr`)

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.

Returns

B
[(multi)digraph-like] A maximum branching.

minimum_branching

minimum_branching(*G*, *attr='weight'*, *default=1*, *preserve_attrs=False*, *partition=None*)

Returns a minimum branching from G.

Parameters

G
[(multi)digraph-like] The graph to be searched.

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`.

preserve_attrs
[bool] If True, preserve the other attributes of the original graph (that are not passed to `attr`)

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.

Returns

B
[(multi)digraph-like] A minimum branching.

maximum_spanning_arborescence

maximum_spanning_arborescence(*G*, *attr='weight'*, *default=1*, *preserve_attrs=False*, *partition=None*)

Returns a maximum spanning arborescence from G.

Parameters

G
[(multi)digraph-like] The graph to be searched.

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`.

preserve_attrs
[bool] If True, preserve the other attributes of the original graph (that are not passed to `attr`)

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.

Returns

B
[(multi)digraph-like] A maximum spanning arborescence.

Raises

NetworkXException
If the graph does not contain a maximum spanning arborescence.

minimum_spanning_arborescence

minimum_spanning_arborescence(*G*, *attr='weight'*, *default=1*, *preserve_attrs=False*, *partition=None*)

Returns a minimum spanning arborescence from G.

Parameters

G
[(multi)digraph-like] The graph to be searched.

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`.

preserve_attrs
[bool] If True, preserve the other attributes of the original graph (that are not passed to `attr`)

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.

Returns

B
[(multi)digraph-like] A minimum spanning arborescence.

Raises

NetworkXException
If the graph does not contain a minimum spanning arborescence.

networkx.algorithms.tree.branchings.ArborescenceIterator

class ArborescenceIterator(*G*, *weight='weight'*, *minimum=True*, *init_partition=None*)

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]

__init__(*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 `(u, v)`-tuple inside an iterable such as a list or set.

Methods

networkx.algorithms.tree.branchings.Edmonds

class Edmonds(*G*, *seed=None*)

Edmonds algorithm [1] for finding optimal branchings and spanning arborescences.

This algorithm can find both minimum and maximum spanning arborescences and branchings.

Notes

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]

__init__(*G*, *seed=None*)

Methods

find_optimum([attr, default, kind, style, ...])	Returns a branching from G.

Edmonds.find_optimum

Edmonds.**find_optimum**(*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 *Randomness*.

Returns

H
[(multi)digraph] The branching.

3.64.3 Encoding and decoding

Functions for encoding and decoding trees.

Since a tree is a highly restricted form of graph, it can be represented concisely in several ways. This module includes functions for encoding and decoding trees in the form of nested tuples and Prüfer sequences. The former requires a rooted tree, whereas the latter can be applied to unrooted trees. Furthermore, there is a bijection from Prüfer sequences to labeled trees.

from_nested_tuple(sequence[, ...])	Returns the rooted tree corresponding to the given nested tuple.
to_nested_tuple(T, root[, canonical_form])	Returns a nested tuple representation of the given tree.
from_prufer_sequence(sequence)	Returns the tree corresponding to the given Prüfer sequence.
to_prufer_sequence(T)	Returns the Prüfer sequence of the given tree.

from_nested_tuple

from_nested_tuple(*sequence*, *sensible_relabeling=False*)

Returns the rooted tree corresponding to the given nested tuple.

The nested tuple representation of a tree is defined recursively. The tree with one node and no edges is represented by the empty tuple, `()`. A tree with `k` subtrees is represented by a tuple of length `k` in which each element is the nested tuple representation of a subtree.

Parameters

sequence
[tuple] A nested tuple representing a rooted tree.

sensible_relabeling
[bool] Whether to relabel the nodes of the tree so that nodes are labeled in increasing order according to their breadth-first search order from the root node.

Returns

NetworkX graph
The tree corresponding to the given nested tuple, whose root node is node 0. If `sensible_labeling` is `True`, nodes will be labeled in breadth-first search order starting from the root node.

See also:

to_nested_tuple
from_prufer_sequence

Notes

This function is *not* the inverse of *to_nested_tuple()*; the only guarantee is that the rooted trees are isomorphic.

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
```

to_nested_tuple

to_nested_tuple(*T*, *root*, *canonical_form=False*)

Returns a nested tuple representation of the given tree.

The nested tuple representation of a tree is defined recursively. The tree with one node and no edges is represented by the empty tuple, `()`. A tree with `k` subtrees is represented by a tuple of length `k` in which each element is the nested tuple representation of a subtree.

Parameters

T
[NetworkX graph] An undirected graph object representing a tree.

root
[node] The node in `T` to interpret as the root of the tree.

canonical_form
[bool] If `True`, each tuple is sorted so that the function returns a canonical form for rooted trees. This means "lighter" subtrees will appear as nested tuples before "heavier" subtrees. In this way, each isomorphic rooted tree has the same nested tuple representation.

Returns

tuple
A nested tuple representation of the tree.

See also:

from_nested_tuple
to_prufer_sequence

Notes

This function is *not* the inverse of *from_nested_tuple()*; the only guarantee is that the rooted trees are isomorphic.

Examples

The tree need not be a balanced binary tree:

```
>>> T = nx.Graph()
>>> T.add_edges_from([(0, 1), (0, 2), (0, 3)])
>>> T.add_edges_from([(1, 4), (1, 5)])
>>> T.add_edges_from([(3, 6), (3, 7)])
>>> root = 0
>>> nx.to_nested_tuple(T, root)
(((), ()), (), (), ())
```

Continuing the above example, if `canonical_form` is `True`, the nested tuples will be sorted:

```
>>> nx.to_nested_tuple(T, root, canonical_form=True)
((), (), (), (), ())
```

Even the path graph can be interpreted as a tree:

```
>>> T = nx.path_graph(4)
>>> root = 0
>>> nx.to_nested_tuple(T, root)
((((),),),)
```

from_prufer_sequence

from_prufer_sequence(*sequence*)

Returns the tree corresponding to the given Prüfer sequence.

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

sequence
[list] A Prüfer sequence, which is a list of *n* - 2 integers between zero and *n* - 1, inclusive.

Returns

NetworkX graph
The tree corresponding to the given Prüfer sequence.

See also:

from_nested_tuple
to_prufer_sequence

Notes

There is a bijection from labeled trees to Prüfer sequences. This function is the inverse of the *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 `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 $O(n)$.

References

[1]

Examples

There is a bijection between Prüfer sequences and labeled trees, so this function is the inverse of the *to_prufer_sequence()* function:

```
>>> edges = [(0, 3), (1, 3), (2, 3), (3, 4), (4, 5)]
>>> tree = nx.Graph(edges)
>>> sequence = nx.to_prufer_sequence(tree)
>>> sequence
[3, 3, 3, 4]
>>> tree2 = nx.from_prufer_sequence(sequence)
>>> list(tree2.edges()) == edges
True
```

to_prufer_sequence

to_prufer_sequence(*T*)

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}.

See also:

to_nested_tuple
from_prufer_sequence

Notes

There is a bijection from labeled trees to Prüfer sequences. This function is the inverse of the *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 `relabel_nodes()` to relabel the nodes of your tree to the appropriate format.

This implementation is from [1] and has a running time of $O(n)$.

References

[1]

Examples

There is a bijection between Prüfer sequences and labeled trees, so this function is the inverse of the *from_prufer_sequence()* function:

```
>>> edges = [(0, 3), (1, 3), (2, 3), (3, 4), (4, 5)]
>>> tree = nx.Graph(edges)
>>> sequence = nx.to_prufer_sequence(tree)
>>> sequence
[3, 3, 3, 4]
>>> tree2 = nx.from_prufer_sequence(sequence)
>>> list(tree2.edges()) == edges
True
```

3.64.4 Operations

Operations on trees.

join(rooted_trees[, label_attribute])	Returns a new rooted tree with a root node joined with the roots of each of the given rooted trees.

join

join(*rooted_trees*, *label_attribute=None*)

Returns a new rooted tree with a root node joined with the roots of each of the given rooted trees.

Parameters

rooted_trees
[list] A list of pairs in which each left element is a NetworkX graph object representing a tree and each right element is the root node of that tree. The nodes of these trees will be relabeled to integers.

label_attribute
[str] If provided, the old node labels will be stored in the new tree under this node attribute. If not provided, the node attribute `'_old'` will store the original label of the node in the rooted trees given in the input.

Returns

NetworkX graph
The rooted tree whose subtrees are the given rooted trees. The new root node is labeled 0. Each non-root node has an attribute, as described under the keyword argument `label_attribute`, that indicates the label of the original node in the input tree.

Notes

Graph, edge, and node attributes are propagated from the given rooted trees to the created tree. If there are any overlapping graph attributes, those from later trees will overwrite those from earlier trees in the tuple of positional arguments.

Examples

Join two full balanced binary trees of height h to get a full balanced binary tree of depth $h + 1$:

```
>>> h = 4
>>> left = nx.balanced_tree(2, h)
>>> right = nx.balanced_tree(2, h)
>>> joined_tree = nx.join([(left, 0), (right, 0)])
>>> nx.is_isomorphic(joined_tree, nx.balanced_tree(2, h + 1))
True
```

3.64.5 Spanning Trees

Algorithms for calculating min/max spanning trees/forests.

minimum_spanning_tree(G[, weight, ...])	Returns a minimum spanning tree or forest on an undirected graph G.
maximum_spanning_tree(G[, weight, ...])	Returns a maximum spanning tree or forest on an undirected graph G.
random_spanning_tree(G[, weight, ...])	Sample a random spanning tree using the edges weights of G.
minimum_spanning_edges(G[, algorithm, ...])	Generate edges in a minimum spanning forest of an undirected weighted graph.
maximum_spanning_edges(G[, algorithm, ...])	Generate edges in a maximum spanning forest of an undirected weighted graph.
SpanningTreeIterator(G[, weight, minimum, ...])	Iterate over all spanning trees of a graph in either increasing or decreasing cost.

minimum_spanning_tree

minimum_spanning_tree(*G*, *weight='weight'*, *algorithm='kruskal'*, *ignore_nan=False*)

Returns a minimum 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 minimum 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 minimum spanning tree or forest.

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 `networkx.tree.recognition` for more detailed definitions.

Isolated nodes with self-loops are in the tree as edgeless isolated nodes.

Examples

```
>>> G = nx.cycle_graph(4)
>>> G.add_edge(0, 3, weight=2)
>>> T = nx.minimum_spanning_tree(G)
>>> sorted(T.edges(data=True))
[(0, 1, {}), (1, 2, {}), (2, 3, {})]
```

maximum_spanning_tree

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.

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 `networkx.tree.recognition` for more detailed definitions.

Isolated nodes with self-loops are in the tree as edgeless isolated nodes.

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, {})]
```

random_spanning_tree

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 verison, 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 *Randomness*.

Returns

nx.Graph
A spanning tree using the distribution defined by the weight of the tree.

References

[1]

minimum_spanning_edges

minimum_spanning_edges(*G*, *algorithm='kruskal'*, *weight='weight'*, *keys=True*, *data=True*, *ignore_nan=False*)

Generate edges in a minimum spanning forest of an undirected weighted graph.

A minimum spanning tree is a subgraph of the graph (a tree) with the minimum sum of edge weights. A spanning forest is a union of the spanning trees for each connected component of the graph.

Parameters

G
[undirected Graph] An undirected graph. If `G` is connected, then the algorithm finds a spanning tree. Otherwise, a spanning forest is found.

algorithm
[string] The algorithm to use when finding a minimum spanning tree. Valid choices are 'kruskal', 'prim', or 'boruvka'. The default is 'kruskal'.

weight
[string] Edge data key to use for weight (default 'weight').

keys
[bool] Whether to yield edge key in multigraphs in addition to the edge. If `G` is not a multigraph, this is ignored.

data
[bool, optional] If True yield the edge data along with the edge.

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

edges
[iterator] An iterator over edges in a maximum spanning tree of `G`. Edges connecting nodes `u` and `v` are represented as tuples: `(u, v, k, d)` or `(u, v, k)` or `(u, v, d)` or `(u, v)`

If `G` is a multigraph, `keys` indicates whether the edge key `k` will be reported in the third position in the edge tuple. `data` indicates whether the edge datadict `d` will appear at the end of the edge tuple.

If `G` is not a multigraph, the tuples are `(u, v, d)` if `data` is True or `(u, v)` if `data` is False.

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.

Modified code from David Eppstein, April 2006 http://www.ics.uci.edu/~eppstein/PADS/

Examples

```
>>> from networkx.algorithms import tree
```

Find minimum spanning edges by Kruskal's algorithm

```
>>> G = nx.cycle_graph(4)
>>> G.add_edge(0, 3, weight=2)
>>> mst = tree.minimum_spanning_edges(G, algorithm="kruskal", data=False)
>>> edgelist = list(mst)
>>> sorted(sorted(e) for e in edgelist)
[[0, 1], [1, 2], [2, 3]]
```

Find minimum spanning edges by Prim's algorithm

```
>>> G = nx.cycle_graph(4)
>>> G.add_edge(0, 3, weight=2)
>>> mst = tree.minimum_spanning_edges(G, algorithm="prim", data=False)
>>> edgelist = list(mst)
>>> sorted(sorted(e) for e in edgelist)
[[0, 1], [1, 2], [2, 3]]
```

maximum_spanning_edges

maximum_spanning_edges(*G*, *algorithm='kruskal'*, *weight='weight'*, *keys=True*, *data=True*, *ignore_nan=False*)

Generate edges in a maximum spanning forest of an undirected weighted graph.

A maximum spanning tree is a subgraph of the graph (a tree) with the maximum possible sum of edge weights. A spanning forest is a union of the spanning trees for each connected component of the graph.

Parameters

G
[undirected Graph] An undirected graph. If `G` is connected, then the algorithm finds a spanning tree. Otherwise, a spanning forest is found.

algorithm
[string] The algorithm to use when finding a maximum spanning tree. Valid choices are 'kruskal', 'prim', or 'boruvka'. The default is 'kruskal'.

weight
[string] Edge data key to use for weight (default 'weight').

keys
[bool] Whether to yield edge key in multigraphs in addition to the edge. If `G` is not a multigraph, this is ignored.

data
[bool, optional] If True yield the edge data along with the edge.

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

edges
[iterator] An iterator over edges in a maximum spanning tree of `G`. Edges connecting nodes `u` and `v` are represented as tuples: `(u, v, k, d)` or `(u, v, k)` or `(u, v, d)` or `(u, v)`

If `G` is a multigraph, `keys` indicates whether the edge key `k` will be reported in the third position in the edge tuple. `data` indicates whether the edge datadict `d` will appear at the end of the edge tuple.

If `G` is not a multigraph, the tuples are `(u, v, d)` if `data` is True or `(u, v)` if `data` is False.

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.

Modified code from David Eppstein, April 2006 http://www.ics.uci.edu/~eppstein/PADS/

Examples

```
>>> from networkx.algorithms import tree
```

Find maximum spanning edges by Kruskal's algorithm

```
>>> G = nx.cycle_graph(4)
>>> G.add_edge(0, 3, weight=2)
>>> mst = tree.maximum_spanning_edges(G, algorithm="kruskal", data=False)
>>> edgelist = list(mst)
>>> sorted(sorted(e) for e in edgelist)
[[0, 1], [0, 3], [1, 2]]
```

Find maximum spanning edges by Prim's algorithm

```
>>> G = nx.cycle_graph(4)
>>> G.add_edge(0, 3, weight=2)  # assign weight 2 to edge 0-3
>>> mst = tree.maximum_spanning_edges(G, algorithm="prim", data=False)
>>> edgelist = list(mst)
>>> sorted(sorted(e) for e in edgelist)
[[0, 1], [0, 3], [2, 3]]
```

networkx.algorithms.tree.mst.SpanningTreeIterator

class SpanningTreeIterator(*G*, *weight='weight'*, *minimum=True*, *ignore_nan=False*)

Iterate over all spanning trees 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) as well as a modified Kruskal's Algorithm to generate minimum spanning trees which respect the partition of edges. For spanning trees with the same weight, ties are broken arbitrarily.

References

[1]

__init__(*G*, *weight='weight'*, *minimum=True*, *ignore_nan=False*)

Initialize the iterator

Parameters

G
[nx.Graph] 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.

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.

Methods

3.64.6 Decomposition

Function for computing a junction tree of a graph.

junction_tree(G)	Returns a junction tree of a given graph.

junction_tree

junction_tree(*G*)

Returns a junction tree of a given graph.

A junction tree (or clique tree) is constructed from a (un)directed graph G. The tree is constructed based on a moralized and triangulated version of G. The tree's nodes consist of maximal cliques and sepsets of the revised graph. The sepset of two cliques is the intersection of the nodes of these cliques, e.g. the sepset of (A,B,C) and (A,C,E,F) is (A,C). These nodes are often called "variables" in this literature. The tree is bipartitie with each sepset connected to its two cliques.

Junction Trees are not unique as the order of clique consideration determines which sepsets are included.

The junction tree algorithm consists of five steps [1]:

1. Moralize the graph
2. Triangulate the graph
3. Find maximal cliques
4. Build the tree from cliques, connecting cliques with shared nodes, set edge-weight to number of shared variables
5. Find maximum spanning tree

Parameters

G
[networkx.Graph] Directed or undirected graph.

Returns

junction_tree
[networkx.Graph] The corresponding junction tree of `G`.

Raises

NetworkXNotImplemented
Raised if `G` is an instance of `MultiGraph` or `MultiDiGraph`.

References

[1], [2]

3.64.7 Exceptions

Functions for encoding and decoding trees.

Since a tree is a highly restricted form of graph, it can be represented concisely in several ways. This module includes functions for encoding and decoding trees in the form of nested tuples and Prüfer sequences. The former requires a rooted tree, whereas the latter can be applied to unrooted trees. Furthermore, there is a bijection from Prüfer sequences to labeled trees.

NotATree	Raised when a function expects a tree (that is, a connected undirected graph with no cycles) but gets a non-tree graph as input instead.

NotATree

exception NotATree

Raised when a function expects a tree (that is, a connected undirected graph with no cycles) but gets a non-tree graph as input instead.

3.65 Triads

Functions for analyzing triads of a graph.

triadic_census(G[, nodelist])	Determines the triadic census of a directed graph.
random_triad(G[, seed])	Returns a random triad from a directed graph.
triads_by_type(G)	Returns a list of all triads for each triad type in a directed graph.
triad_type(G)	Returns the sociological triad type for a triad.
is_triad(G)	Returns True if the graph G is a triad, else False.
all_triads(G)	A generator of all possible triads in G.
all_triplets(G)	Returns a generator of all possible sets of 3 nodes in a DiGraph.

3.65.1 triadic_census

triadic_census(*G*, *nodelist=None*)

Determines the triadic census of a directed graph.

The triadic census is a count of how many of the 16 possible types of triads are present in a directed graph. If a list of nodes is passed, then only those triads are taken into account which have elements of nodelist in them.

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.

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`

Notes

This algorithm has complexity $O(m)$ where m is the number of edges in the graph.

References

[1]

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
```

3.65.2 random_triad

random_triad(*G*, *seed=None*)

Returns a random triad from a directed graph.

Parameters

G
[digraph] A NetworkX DiGraph

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

G2
[subgraph] A randomly selected triad (order-3 NetworkX DiGraph)

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)])
```

3.65.3 triads_by_type

triads_by_type(*G*)

Returns a list of all triads for each triad type in a directed graph. There are exactly 16 different types of triads possible. Suppose 1, 2, 3 are three nodes, they will be classified as a particular triad type if their connections are as follows:

- 003: 1, 2, 3
- 012: 1 -> 2, 3
- 102: 1 <-> 2, 3
- 021D: 1 <- 2 -> 3
- 021U: 1 -> 2 <- 3
- 021C: 1 -> 2 -> 3
- 111D: 1 <-> 2 <- 3
- 111U: 1 <-> 2 -> 3
- 030T: 1 -> 2 -> 3, 1 -> 3
- 030C: 1 <- 2 <- 3, 1 -> 3
- 201: 1 <-> 2 <-> 3
- 120D: 1 <- 2 -> 3, 1 <-> 3
- 120U: 1 -> 2 <- 3, 1 <-> 3
- 120C: 1 -> 2 -> 3, 1 <-> 3

- 210: 1 -> 2 <-> 3, 1 <-> 3
- 300: 1 <-> 2 <-> 3, 1 <-> 3

Refer to the example gallery for visual examples of the triad types.

Parameters

G
[digraph] A NetworkX DiGraph

Returns

tri_by_type
[dict] Dictionary with triad types as keys and lists of triads as values.

References

[1]

Examples

```
>>> G = nx.DiGraph([(1, 2), (1, 3), (2, 3), (3, 1), (5, 6), (5, 4), (6, 7)])
>>> dict = nx.triads_by_type(G)
>>> dict['120C'][0].edges()
OutEdgeView([(1, 2), (1, 3), (2, 3), (3, 1)])
>>> dict['012'][0].edges()
OutEdgeView([(1, 2)])
```

3.65.4 triad_type

triad_type(*G*)

Returns the sociological triad type for a triad.

Parameters

G
[digraph] A NetworkX DiGraph with 3 nodes

Returns

triad_type
[str] A string identifying the triad type

Notes

There can be 6 unique edges in a triad (order-3 DiGraph) (so 2^^6=64 unique triads given 3 nodes). These 64 triads each display exactly 1 of 16 topologies of triads (topologies can be permuted). These topologies are identified by the following notation:

{m}{a}{n}{type} (for example: 111D, 210, 102)

Here:

{m} = number of mutual ties (takes 0, 1, 2, 3); a mutual tie is (0,1)
AND (1,0)

{a} = number of assymmetric ties (takes 0, 1, 2, 3); an assymmetric tie
is (0,1) BUT NOT (1,0) or vice versa

{n} = number of null ties (takes 0, 1, 2, 3); a null tie is NEITHER
(0,1) NOR (1,0)

{type} = a letter (takes U, D, C, T) corresponding to up, down, cyclical
and transitive. This is only used for topologies that can have more than one form (eg: 021D and 021U).

References

[1]

Examples

```
>>> G = nx.DiGraph([(1, 2), (2, 3), (3, 1)])
>>> nx.triad_type(G)
'030C'
>>> G.add_edge(1, 3)
>>> nx.triad_type(G)
'120C'
```

3.65.5 is_triad

is_triad(*G*)

Returns True if the graph G is a triad, else False.

Parameters

G
[graph] A NetworkX Graph

Returns

istriad
[boolean] Whether G is a valid triad

Examples

```
>>> G = nx.DiGraph([(1, 2), (2, 3), (3, 1)])
>>> nx.is_triad(G)
True
>>> G.add_edge(0, 1)
>>> nx.is_triad(G)
False
```

3.65.6 all_triads

all_triads(*G*)

A generator of all possible triads in G.

Parameters

G
[digraph] A NetworkX DiGraph

Returns

all_triads
[generator of DiGraphs] Generator of triads (order-3 DiGraphs)

Examples

```
>>> G = nx.DiGraph([(1, 2), (2, 3), (3, 1), (3, 4), (4, 1), (4, 2)])
>>> for triad in nx.all_triads(G):
...     print(triad.edges)
[(1, 2), (2, 3), (3, 1)]
[(1, 2), (4, 1), (4, 2)]
[(3, 1), (3, 4), (4, 1)]
[(2, 3), (3, 4), (4, 2)]
```

3.65.7 all_triplets

all_triplets(*G*)

Returns a generator of all possible sets of 3 nodes in a DiGraph.

Parameters

G
[digraph] A NetworkX DiGraph

Returns

triplets
[generator of 3-tuples] Generator of tuples of 3 nodes

Examples

```
>>> G = nx.DiGraph([(1, 2), (2, 3), (3, 4)])
>>> list(nx.all_triplets(G))
[(1, 2, 3), (1, 2, 4), (1, 3, 4), (2, 3, 4)]
```

3.66 Vitality

Vitality measures.

closeness_vitality(G[, node, weight, ...])	Returns the closeness vitality for nodes in the graph.

3.66.1 closeness_vitality

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.

weight
[string] The name of the edge attribute used as weight. This is passed directly to the `wiener_index()` function.

node
[object] If specified, only the closeness vitality for this node will be returned. Otherwise, a dictionary mapping each node to its closeness vitality will be returned.

Returns

dictionary or float
If `node` is None, this function returns a dictionary with nodes as keys and closeness vitality as the value. Otherwise, it returns only the closeness vitality for the specified `node`.

The closeness vitality of a node may be negative infinity if removing that node would disconnect the graph.

Other Parameters

wiener_index
[number] If you have already computed the Wiener index of the graph `G`, you can provide that value here. Otherwise, it will be computed for you.

See also:

closeness_centrality

References

[1]

Examples

```
>>> G = nx.cycle_graph(3)
>>> nx.closeness_vitality(G)
{0: 2.0, 1: 2.0, 2: 2.0}
```

3.67 Voronoi cells

Functions for computing the Voronoi cells of a graph.

voronoi_cells(G, center_nodes[, weight])	Returns the Voronoi cells centered at `center_nodes` with respect to the shortest-path distance metric.

3.67.1 voronoi_cells

voronoi_cells(*G*, *center_nodes*, *weight='weight'*)

Returns the Voronoi cells centered at `center_nodes` with respect to the shortest-path distance metric.

If *C* is a set of nodes in the graph and *c* is an element of *C*, the *Voronoi cell* centered at a node *c* is the set of all nodes *v* that are closer to *c* than to any other center node in *C* with respect to the shortest-path distance metric. [1]

For directed graphs, this will compute the "outward" Voronoi cells, as defined in [1], in which distance is measured from the center nodes to the target node. For the "inward" Voronoi cells, use the `DiGraph.reverse()` method to reverse the orientation of the edges before invoking this function on the directed graph.

Parameters

G
[NetworkX graph]

center_nodes
[set] A nonempty set of nodes in the graph `G` that represent the center of the Voronoi cells.

weight
[string or function] The edge attribute (or an arbitrary function) representing the weight of an edge. This keyword argument is as described in the documentation for `multi_source_dijkstra_path()`, for example.

Returns

dictionary
A mapping from center node to set of all nodes in the graph closer to that center node than to any other center node. The keys of the dictionary are the element of `center_nodes`, and the values of the dictionary form a partition of the nodes of `G`.

Raises

ValueError
If `center_nodes` is empty.

References

[1]

Examples

To get only the partition of the graph induced by the Voronoi cells, take the collection of all values in the returned dictionary:

```
>>> G = nx.path_graph(6)
>>> center_nodes = {0, 3}
>>> cells = nx.voronoi_cells(G, center_nodes)
>>> partition = set(map(frozenset, cells.values()))
>>> sorted(map(sorted, partition))
[[0, 1], [2, 3, 4, 5]]
```

3.68 Wiener index

Functions related to the Wiener index of a graph.

wiener_index(G[, weight])	Returns the Wiener index of the given graph.

3.68.1 wiener_index

wiener_index(*G*, *weight=None*)

Returns the Wiener index of the given graph.

The *Wiener index* of a graph is the sum of the shortest-path distances between each pair of reachable nodes. For pairs of nodes in undirected graphs, only one orientation of the pair is counted.

Parameters

G
[NetworkX graph]

weight
[object] The edge attribute to use as distance when computing shortest-path distances. This is passed directly to the `networkx.shortest_path_length()` function.

Returns

float
The Wiener index of the graph `G`.

Raises

NetworkXError
If the graph `G` is not connected.

Notes

If a pair of nodes is not reachable, the distance is assumed to be infinity. This means that for graphs that are not strongly-connected, this function returns `inf`.

The Wiener index is not usually defined for directed graphs, however this function uses the natural generalization of the Wiener index to directed graphs.

Examples

The Wiener index of the (unweighted) complete graph on *n* nodes equals the number of pairs of the *n* nodes, since each pair of nodes is at distance one:

```
>>> n = 10
>>> G = nx.complete_graph(n)
>>> nx.wiener_index(G) == n * (n - 1) / 2
True
```

Graphs that are not strongly-connected have infinite Wiener index:

```
>>> G = nx.empty_graph(2)
>>> nx.wiener_index(G)
inf
```

CHAPTER

FOUR

FUNCTIONS

Functional interface to graph methods and assorted utilities.

4.1 Graph

degree(G[, nbunch, weight])	Returns a degree view of single node or of nbunch of nodes.
degree_histogram(G)	Returns a list of the frequency of each degree value.
density(G)	Returns the density of a graph.
create_empty_copy(G[, with_data])	Returns a copy of the graph G with all of the edges removed.
is_directed(G)	Return True if graph is directed.
to_directed(graph)	Returns a directed view of the graph `graph`.
to_undirected(graph)	Returns an undirected view of the graph `graph`.
is_empty(G)	Returns True if `G` has no edges.
add_star(G_to_add_to, nodes_for_star, **attr)	Add a star to Graph G_to_add_to.
add_path(G_to_add_to, nodes_for_path, **attr)	Add a path to the Graph G_to_add_to.
add_cycle(G_to_add_to, nodes_for_cycle, **attr)	Add a cycle to the Graph G_to_add_to.
subgraph(G, nbunch)	Returns the subgraph induced on nodes in nbunch.
subgraph_view(G[, filter_node, filter_edge])	View of `G` applying a filter on nodes and edges.
induced_subgraph(G, nbunch)	Returns a SubGraph view of `G` showing only nodes in nbunch.
restricted_view(G, nodes, edges)	Returns a view of `G` with hidden nodes and edges.
reverse_view(G)	View of `G` with edge directions reversed
edge_subgraph(G, edges)	Returns a view of the subgraph induced by the specified edges.

4.1.1 degree

degree(*G*, *nbunch=None*, *weight=None*)

Returns a degree view of single node or of nbunch of nodes. If nbunch is omitted, then return degrees of *all* nodes.

4.1.2 degree_histogram

degree_histogram(*G*)

Returns a list of the frequency of each degree value.

Parameters

G
[Networkx graph] A graph

Returns

hist
[list] A list of frequencies of degrees. The degree values are the index in the list.

Notes

Note: the bins are width one, hence len(list) can be large (Order(number_of_edges))

4.1.3 density

density(*G*)

Returns the density of a graph.

The density for undirected graphs is

$$d = \frac{2m}{n(n-1)},$$

and for directed graphs is

$$d = \frac{m}{n(n-1)},$$

where `n` is the number of nodes and `m` is the number of edges in `G`.

Notes

The density is 0 for a graph without edges and 1 for a complete graph. The density of multigraphs can be higher than 1.

Self loops are counted in the total number of edges so graphs with self loops can have density higher than 1.

4.1.4 create_empty_copy

create_empty_copy(*G*, *with_data=True*)

Returns a copy of the graph G with all of the edges removed.

Parameters

G
[graph] A NetworkX graph

with_data
[bool (default=True)] Propagate Graph and Nodes data to the new graph.

See also:

empty_graph

4.1.5 is_directed

is_directed(*G*)

Return True if graph is directed.

4.1.6 to_directed

to_directed(*graph*)

Returns a directed view of the graph `graph`.

Identical to graph.to_directed(as_view=True) Note that graph.to_directed defaults to `as_view=False` while this function always provides a view.

4.1.7 to_undirected

to_undirected(*graph*)

Returns an undirected view of the graph `graph`.

Identical to graph.to_undirected(as_view=True) Note that graph.to_undirected defaults to `as_view=False` while this function always provides a view.

4.1.8 is_empty

is_empty(*G*)

Returns True if `G` has no edges.

Parameters

G
[graph] A NetworkX graph.

Returns

bool
True if `G` has no edges, and False otherwise.

Notes

An empty graph can have nodes but not edges. The empty graph with zero nodes is known as the null graph. This is an $O(n)$ operation where n is the number of nodes in the graph.

4.1.9 add_star

add_star(*G_to_add_to*, *nodes_for_star*, ***attr*)

Add a star to Graph G_to_add_to.

The first node in `nodes_for_star` is the middle of the star. It is connected to all other nodes.

Parameters

G_to_add_to
[graph] A NetworkX graph

nodes_for_star
[iterable container] A container of nodes.

attr
[keyword arguments, optional (default= no attributes)] Attributes to add to every edge in star.

See also:

add_path, ***add_cycle***

Examples

```
>>> G = nx.Graph()
>>> nx.add_star(G, [0, 1, 2, 3])
>>> nx.add_star(G, [10, 11, 12], weight=2)
```

4.1.10 add_path

add_path(*G_to_add_to*, *nodes_for_path*, ***attr*)

Add a path to the Graph G_to_add_to.

Parameters

G_to_add_to
[graph] A NetworkX graph

nodes_for_path
[iterable container] A container of nodes. A path will be constructed from the nodes (in order) and added to the graph.

attr
[keyword arguments, optional (default= no attributes)] Attributes to add to every edge in path.

See also:

add_star, ***add_cycle***

Examples

```
>>> G = nx.Graph()
>>> nx.add_path(G, [0, 1, 2, 3])
>>> nx.add_path(G, [10, 11, 12], weight=7)
```

4.1.11 add_cycle

add_cycle(*G_to_add_to*, *nodes_for_cycle*, ***attr*)

Add a cycle to the Graph G_to_add_to.

Parameters

G_to_add_to
[graph] A NetworkX graph

nodes_for_cycle: iterable container
A container of nodes. A cycle will be constructed from the nodes (in order) and added to the graph.

attr
[keyword arguments, optional (default= no attributes)] Attributes to add to every edge in cycle.

See also:

add_path*, *add_star

Examples

```
>>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> nx.add_cycle(G, [0, 1, 2, 3])
>>> nx.add_cycle(G, [10, 11, 12], weight=7)
```

4.1.12 subgraph

subgraph(*G*, *nbunch*)

Returns the subgraph induced on nodes in nbunch.

Parameters

G
[graph] A NetworkX graph

nbunch
[list, iterable] A container of nodes that will be iterated through once (thus it should be an iterator or be iterable). Each element of the container should be a valid node type: any hashable type except None. If nbunch is None, return all edges data in the graph. Nodes in nbunch that are not in the graph will be (quietly) ignored.

Notes

subgraph(G) calls G.subgraph()

4.1.13 subgraph_view

subgraph_view(*G*, *filter_node=<function no_filter>*, *filter_edge=<function no_filter>*)

View of `G` applying a filter on nodes and edges.

`subgraph_view` provides a read-only view of the input graph that excludes nodes and edges based on the outcome of two filter functions `filter_node` and `filter_edge`.

The `filter_node` function takes one argument — the node — and returns `True` if the node should be included in the subgraph, and `False` if it should not be included.

The `filter_edge` function takes two (or three arguments if `G` is a multi-graph) — the nodes describing an edge, plus the edge-key if parallel edges are possible — and returns `True` if the edge should be included in the subgraph, and `False` if it should not be included.

Both node and edge filter functions are called on graph elements as they are queried, meaning there is no up-front cost to creating the view.

Parameters

G
: [networkx.Graph] A directed/undirected graph/multigraph

filter_node
: [callable, optional] A function taking a node as input, which returns `True` if the node should appear in the view.

filter_edge
: [callable, optional] A function taking as input the two nodes describing an edge (plus the edge-key if `G` is a multi-graph), which returns `True` if the edge should appear in the view.

Returns

graph
: [networkx.Graph] A read-only graph view of the input graph.

Examples

```
>>> G = nx.path_graph(6)
```

Filter functions operate on the node, and return `True` if the node should appear in the view:

```
>>> def filter_node(n1):
...     return n1 != 5
...
>>> view = nx.subgraph_view(G, filter_node=filter_node)
>>> view.nodes()
NodeView((0, 1, 2, 3, 4))
```

We can use a closure pattern to filter graph elements based on additional data — for example, filtering on edge data attached to the graph:

```
>>> G[3][4]["cross_me"] = False
>>> def filter_edge(n1, n2):
...     return G[n1][n2].get("cross_me", True)
...
>>> view = nx.subgraph_view(G, filter_edge=filter_edge)
>>> view.edges()
EdgeView([(0, 1), (1, 2), (2, 3), (4, 5)])
```

```
>>> view = nx.subgraph_view(G, filter_node=filter_node, filter_edge=filter_edge,)
>>> view.nodes()
NodeView((0, 1, 2, 3, 4))
>>> view.edges()
EdgeView([(0, 1), (1, 2), (2, 3)])
```

4.1.14 induced_subgraph

induced_subgraph(*G*, *nbunch*)

Returns a SubGraph view of `G` showing only nodes in nbunch.

The induced subgraph of a graph on a set of nodes N is the graph with nodes N and edges from G which have both ends in N.

Parameters

G
[NetworkX Graph]

nbunch
[node, container of nodes or None (for all nodes)]

Returns

subgraph
[SubGraph View] A read-only view of the subgraph in `G` induced by the nodes. Changes to the graph `G` will be reflected in the view.

Notes

To create a mutable subgraph with its own copies of nodes edges and attributes use `subgraph.copy()` or `Graph(subgraph)`

For an inplace reduction of a graph to a subgraph you can remove nodes: `G.remove_nodes_from(n in G if n not in set(nbunch))`

If you are going to compute subgraphs of your subgraphs you could end up with a chain of views that can be very slow once the chain has about 15 views in it. If they are all induced subgraphs, you can short-cut the chain by making them all subgraphs of the original graph. The graph class method `G.subgraph` does this when `G` is a subgraph. In contrast, this function allows you to choose to build chains or not, as you wish. The returned subgraph is a view on `G`.

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> H = nx.induced_subgraph(G, [0, 1, 3])
>>> list(H.edges)
[(0, 1)]
>>> list(H.nodes)
[0, 1, 3]
```

4.1.15 restricted_view

restricted_view(*G*, *nodes*, *edges*)

Returns a view of `G` with hidden nodes and edges.

The resulting subgraph filters out node *nodes* and edges *edges*. Filtered out nodes also filter out any of their edges.

Parameters

G
[NetworkX Graph]

nodes
[iterable] An iterable of nodes. Nodes not present in `G` are ignored.

edges
[iterable] An iterable of edges. Edges not present in `G` are ignored.

Returns

subgraph
[SubGraph View] A read-only restricted view of `G` filtering out nodes and edges. Changes to `G` are reflected in the view.

Notes

To create a mutable subgraph with its own copies of nodes edges and attributes use `subgraph.copy()` or `Graph(subgraph)`

If you create a subgraph of a subgraph recursively you may end up with a chain of subgraph views. Such chains can get quite slow for lengths near 15. To avoid long chains, try to make your subgraph based on the original graph. We do not rule out chains programmatically so that odd cases like an *edge_subgraph* of a *restricted_view* can be created.

Examples

```
>>> G = nx.path_graph(5)
>>> H = nx.restricted_view(G, [0], [(1, 2), (3, 4)])
>>> list(H.nodes)
[1, 2, 3, 4]
>>> list(H.edges)
[(2, 3)]
```

4.1.16 reverse_view

reverse_view(*G*)

View of G with edge directions reversed

reverse_view returns a read-only view of the input graph where edge directions are reversed.

Identical to digraph.reverse(copy=False)

Parameters

G
[networkx.DiGraph]

Returns

graph
[networkx.DiGraph]

Examples

```
>>> G = nx.DiGraph()
>>> G.add_edge(1, 2)
>>> G.add_edge(2, 3)
>>> G.edges()
OutEdgeView([(1, 2), (2, 3)])
```

```
>>> view = nx.reverse_view(G)
>>> view.edges()
OutEdgeView([(2, 1), (3, 2)])
```

4.1.17 edge_subgraph

edge_subgraph(*G*, *edges*)

Returns a view of the subgraph induced by the specified edges.

The induced subgraph contains each edge in *edges* and each node incident to any of those edges.

Parameters

G
[NetworkX Graph]

edges
[iterable] An iterable of edges. Edges not present in G are ignored.

Returns

subgraph
[SubGraph View] A read-only edge-induced subgraph of G. Changes to G are reflected in the view.

Notes

To create a mutable subgraph with its own copies of nodes edges and attributes use `subgraph.copy()` or `Graph(subgraph)`

If you create a subgraph of a subgraph recursively you can end up with a chain of subgraphs that becomes very slow with about 15 nested subgraph views. Luckily the edge_subgraph filter nests nicely so you can use the original graph as G in this function to avoid chains. We do not rule out chains programmatically so that odd cases like an *edge_subgraph* of a *restricted_view* can be created.

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)]
```

4.2 Nodes

nodes(G)	Returns an iterator over the graph nodes.
number_of_nodes(G)	Returns the number of nodes in the graph.
neighbors(G, n)	Returns a list of nodes connected to node n.
all_neighbors(graph, node)	Returns all of the neighbors of a node in the graph.
non_neighbors(graph, node)	Returns the non-neighbors of the node in the graph.
common_neighbors(G, u, v)	Returns the common neighbors of two nodes in a graph.

4.2.1 nodes

nodes(*G*)

Returns an iterator over the graph nodes.

4.2.2 number_of_nodes

number_of_nodes(*G*)

Returns the number of nodes in the graph.

4.2.3 neighbors

neighbors(*G*, *n*)

Returns a list of nodes connected to node n.

4.2.4 all_neighbors

all_neighbors(*graph*, *node*)

Returns all of the neighbors of a node in the graph.

If the graph is directed returns predecessors as well as successors.

Parameters

graph
[NetworkX graph] Graph to find neighbors.

node
[node] The node whose neighbors will be returned.

Returns

neighbors
[iterator] Iterator of neighbors

4.2.5 non_neighbors

non_neighbors(*graph*, *node*)

Returns the non-neighbors of the node in the graph.

Parameters

graph
[NetworkX graph] Graph to find neighbors.

node
[node] The node whose neighbors will be returned.

Returns

non_neighbors
[iterator] Iterator of nodes in the graph that are not neighbors of the node.

4.2.6 common_neighbors

common_neighbors(*G*, *u*, *v*)

Returns the common neighbors of two nodes in a graph.

Parameters

G
[graph] A NetworkX undirected graph.

u, v
[nodes] Nodes in the graph.

Returns

cnbors
[iterator] Iterator of common neighbors of u and v in the graph.

Raises

NetworkXError
If u or v is not a node in the graph.

Examples

```
>>> G = nx.complete_graph(5)
>>> sorted(nx.common_neighbors(G, 0, 1))
[2, 3, 4]
```

4.3 Edges

edges(G[, nbunch])	Returns an edge view of edges incident to nodes in nbunch.
number_of_edges(G)	Returns the number of edges in the graph.
density(G)	Returns the density of a graph.
non_edges(graph)	Returns the non-existent edges in the graph.

4.3.1 edges

edges(*G*, *nbunch=None*)

Returns an edge view of edges incident to nodes in nbunch.

Return all edges if nbunch is unspecified or nbunch=None.

For digraphs, edges=out_edges

4.3.2 number_of_edges

number_of_edges(*G*)

Returns the number of edges in the graph.

4.3.3 non_edges

non_edges(*graph*)

Returns the non-existent edges in the graph.

Parameters

graph
[NetworkX graph.] Graph to find non-existent edges.

Returns

non_edges
[iterator] Iterator of edges that are not in the graph.

4.4 Self loops

selfloop_edges(G[, data, keys, default])	Returns an iterator over selfloop edges.
number_of_selfloops(G)	Returns the number of selfloop edges.
nodes_with_selfloops(G)	Returns an iterator over nodes with self loops.

4.4.1 selfloop_edges

selfloop_edges(*G*, *data=False*, *keys=False*, *default=None*)

Returns an iterator over selfloop edges.

A selfloop edge has the same node at both ends.

Parameters

G
[graph] A NetworkX graph.

data
[string or bool, optional (default=False)] Return selfloop edges as two tuples (u, v) (data=False) or three-tuples (u, v, datadict) (data=True) or three-tuples (u, v, datavalue) (data='attrname')

keys
[bool, optional (default=False)] If True, return edge keys with each edge.

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

edgeiter
[iterator over edge tuples] An iterator over all selfloop edges.

See also:

nodes_with_selfloops*, *number_of_selfloops

Examples

```
>>> G = nx.MultiGraph()  # or Graph, DiGraph, MultiDiGraph, etc
>>> ekey = G.add_edge(1, 1)
>>> ekey = G.add_edge(1, 2)
>>> list(nx.selfloop_edges(G))
[(1, 1)]
>>> list(nx.selfloop_edges(G, data=True))
[(1, 1, {})]
>>> list(nx.selfloop_edges(G, keys=True))
[(1, 1, 0)]
>>> list(nx.selfloop_edges(G, keys=True, data=True))
[(1, 1, 0, {})]
```

4.4.2 number_of_selfloops

number_of_selfloops(*G*)

Returns the number of selfloop edges.

A selfloop edge has the same node at both ends.

Returns

nloops
[int] The number of selfloops.

See also:

nodes_with_selfloops, *selfloop_edges*

Examples

```
>>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge(1, 1)
>>> G.add_edge(1, 2)
>>> nx.number_of_selfloops(G)
1
```

4.4.3 nodes_with_selfloops

nodes_with_selfloops(*G*)

Returns an iterator over nodes with self loops.

A node with a self loop has an edge with both ends adjacent to that node.

Returns

nodelist
[iterator] A iterator over nodes with self loops.

See also:

selfloop_edges, *number_of_selfloops*

Examples

```
>>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge(1, 1)
>>> G.add_edge(1, 2)
>>> list(nx.nodes_with_selfloops(G))
[1]
```

4.5 Attributes

is_weighted(G[, edge, weight])	Returns True if G has weighted edges.
is_negatively_weighted(G[, edge, weight])	Returns True if G has negatively weighted edges.
set_node_attributes(G, values[, name])	Sets node attributes from a given value or dictionary of values.
get_node_attributes(G, name)	Get node attributes from graph
set_edge_attributes(G, values[, name])	Sets edge attributes from a given value or dictionary of values.
get_edge_attributes(G, name)	Get edge attributes from graph

4.5.1 is_weighted

is_weighted(*G*, *edge=None*, *weight='weight'*)

Returns True if G has weighted edges.

Parameters

G
[graph] A NetworkX graph.

edge
[tuple, optional] A 2-tuple specifying the only edge in G that will be tested. If None, then every edge in G is tested.

weight: string, optional
The attribute name used to query for edge weights.

Returns

bool
A boolean signifying if G, or the specified edge, is weighted.

Raises

NetworkXError
If the specified edge does not exist.

Examples

```
>>> G = nx.path_graph(4)
>>> nx.is_weighted(G)
False
>>> nx.is_weighted(G, (2, 3))
False
```

```
>>> G = nx.DiGraph()
>>> G.add_edge(1, 2, weight=1)
>>> nx.is_weighted(G)
True
```

4.5.2 is_negatively_weighted

is_negatively_weighted(*G*, *edge=None*, *weight='weight'*)

Returns True if `G` has negatively weighted edges.

Parameters

G
[graph] A NetworkX graph.

edge
[tuple, optional] A 2-tuple specifying the only edge in `G` that will be tested. If None, then every edge in `G` is tested.

weight: string, optional
The attribute name used to query for edge weights.

Returns

bool
A boolean signifying if `G`, or the specified edge, is negatively weighted.

Raises

NetworkXError
If the specified edge does not exist.

Examples

```
>>> G = nx.Graph()
>>> G.add_edges_from([(1, 3), (2, 4), (2, 6)])
>>> G.add_edge(1, 2, weight=4)
>>> nx.is_negatively_weighted(G, (1, 2))
False
>>> G[2][4]["weight"] = -2
>>> nx.is_negatively_weighted(G)
True
>>> G = nx.DiGraph()
>>> edges = [("0", "3", 3), ("0", "1", -5), ("1", "0", -2)]
>>> G.add_weighted_edges_from(edges)
>>> nx.is_negatively_weighted(G)
True
```

4.5.3 set_node_attributes

set_node_attributes(*G*, *values*, *name=None*)

Sets node attributes from a given value or dictionary of values.

> **Warning:** The call order of arguments `values` and `name` switched between v1.x & v2.x.

Parameters

G
[NetworkX Graph]

values
: [scalar value, dict-like] What the node attribute should be set to. If `values` is not a dictionary, then it is treated as a single attribute value that is then applied to every node in `G`. This means that if you provide a mutable object, like a list, updates to that object will be reflected in the node attribute for every node. The attribute name will be `name`.

  If `values` is a dict or a dict of dict, it should be keyed by node to either an attribute value or a dict of attribute key/value pairs used to update the node's attributes.

name
: [string (optional, default=None)] Name of the node attribute to set if values is a scalar.

Examples

After computing some property of the nodes of a graph, you may want to assign a node attribute to store the value of that property for each node:

```
>>> G = nx.path_graph(3)
>>> bb = nx.betweenness_centrality(G)
>>> isinstance(bb, dict)
True
>>> nx.set_node_attributes(G, bb, "betweenness")
>>> G.nodes[1]["betweenness"]
1.0
```

If you provide a list as the second argument, updates to the list will be reflected in the node attribute for each node:

```
>>> G = nx.path_graph(3)
>>> labels = []
>>> nx.set_node_attributes(G, labels, "labels")
>>> labels.append("foo")
>>> G.nodes[0]["labels"]
['foo']
>>> G.nodes[1]["labels"]
['foo']
>>> G.nodes[2]["labels"]
['foo']
```

If you provide a dictionary of dictionaries as the second argument, the outer dictionary is assumed to be keyed by node to an inner dictionary of node attributes for that node:

```
>>> G = nx.path_graph(3)
>>> attrs = {0: {"attr1": 20, "attr2": "nothing"}, 1: {"attr2": 3}}
>>> nx.set_node_attributes(G, attrs)
>>> G.nodes[0]["attr1"]
20
>>> G.nodes[0]["attr2"]
'nothing'
>>> G.nodes[1]["attr2"]
3
>>> G.nodes[2]
{}
```

Note that if the dictionary contains nodes that are not in `G`, the values are silently ignored:

```
>>> G = nx.Graph()
>>> G.add_node(0)
```

(continues on next page)

(continued from previous page)

```
>>> nx.set_node_attributes(G, {0: "red", 1: "blue"}, name="color")
>>> G.nodes[0]["color"]
'red'
>>> 1 in G.nodes
False
```

4.5.4 get_node_attributes

get_node_attributes(*G*, *name*)

Get node attributes from graph

Parameters

G
[NetworkX Graph]

name
[string] Attribute name

Returns

Dictionary of attributes keyed by node.

Examples

```
>>> G = nx.Graph()
>>> G.add_nodes_from([1, 2, 3], color="red")
>>> color = nx.get_node_attributes(G, "color")
>>> color[1]
'red'
```

4.5.5 set_edge_attributes

set_edge_attributes(*G*, *values*, *name=None*)

Sets edge attributes from a given value or dictionary of values.

Warning: The call order of arguments `values` and `name` switched between v1.x & v2.x.

Parameters

G
[NetworkX Graph]

values
[scalar value, dict-like] What the edge attribute should be set to. If `values` is not a dictionary, then it is treated as a single attribute value that is then applied to every edge in `G`. This means that if you provide a mutable object, like a list, updates to that object will be reflected in the edge attribute for each edge. The attribute name will be `name`.

If `values` is a dict or a dict of dict, it should be keyed by edge tuple to either an attribute value or a dict of attribute key/value pairs used to update the edge's attributes. For multigraphs, the

edge tuples must be of the form `(u, v, key)`, where `u` and `v` are nodes and `key` is the edge key. For non-multigraphs, the keys must be tuples of the form `(u, v)`.

name
[string (optional, default=None)] Name of the edge attribute to set if values is a scalar.

Examples

After computing some property of the edges of a graph, you may want to assign a edge attribute to store the value of that property for each edge:

```
>>> G = nx.path_graph(3)
>>> bb = nx.edge_betweenness_centrality(G, normalized=False)
>>> nx.set_edge_attributes(G, bb, "betweenness")
>>> G.edges[1, 2]["betweenness"]
2.0
```

If you provide a list as the second argument, updates to the list will be reflected in the edge attribute for each edge:

```
>>> labels = []
>>> nx.set_edge_attributes(G, labels, "labels")
>>> labels.append("foo")
>>> G.edges[0, 1]["labels"]
['foo']
>>> G.edges[1, 2]["labels"]
['foo']
```

If you provide a dictionary of dictionaries as the second argument, the entire dictionary will be used to update edge attributes:

```
>>> G = nx.path_graph(3)
>>> attrs = {(0, 1): {"attr1": 20, "attr2": "nothing"}, (1, 2): {"attr2": 3}}
>>> nx.set_edge_attributes(G, attrs)
>>> G[0][1]["attr1"]
20
>>> G[0][1]["attr2"]
'nothing'
>>> G[1][2]["attr2"]
3
```

The attributes of one Graph can be used to set those of another.

```
>>> H = nx.path_graph(3)
>>> nx.set_edge_attributes(H, G.edges)
```

Note that if the dict contains edges that are not in `G`, they are silently ignored:

```
>>> G = nx.Graph([(0, 1)])
>>> nx.set_edge_attributes(G, {(1, 2): {"weight": 2.0}})
>>> (1, 2) in G.edges()
False
```

For multigraphs, the `values` dict is expected to be keyed by 3-tuples including the edge key:

```
>>> MG = nx.MultiGraph()
>>> edges = [(0, 1), (0, 1)]
>>> MG.add_edges_from(edges)  # Returns list of edge keys
```

(continues on next page)

(continued from previous page)

```
[0, 1]
>>> attributes = {(0, 1, 0): {"cost": 21}, (0, 1, 1): {"cost": 7}}
>>> nx.set_edge_attributes(MG, attributes)
>>> MG[0][1][0]["cost"]
21
>>> MG[0][1][1]["cost"]
7
```

If MultiGraph attributes are desired for a Graph, you must convert the 3-tuple multiedge to a 2-tuple edge and the last multiedge's attribute value will overwrite the previous values. Continuing from the previous case we get:

```
>>> H = nx.path_graph([0, 1, 2])
>>> nx.set_edge_attributes(H, {(u, v): ed for u, v, ed in MG.edges.data()})
>>> nx.get_edge_attributes(H, "cost")
{(0, 1): 7}
```

4.5.6 get_edge_attributes

get_edge_attributes(*G*, *name*)

Get edge attributes from graph

Parameters

G
[NetworkX Graph]

name
[string] Attribute name

Returns

Dictionary of attributes keyed by edge. For (di)graphs, the keys are 2-tuples of the form: (u, v). For multi(di)graphs, the keys are 3-tuples of the form: (u, v, key).

Examples

```
>>> G = nx.Graph()
>>> nx.add_path(G, [1, 2, 3], color="red")
>>> color = nx.get_edge_attributes(G, "color")
>>> color[(1, 2)]
'red'
```

4.6 Paths

is_path(G, path)	Returns whether or not the specified path exists.
path_weight(G, path, weight)	Returns total cost associated with specified path and weight

4.6.1 is_path

is_path(*G*, *path*)

Returns whether or not the specified path exists.

For it to return True, every node on the path must exist and each consecutive pair must be connected via one or more edges.

Parameters

G
[graph] A NetworkX graph.

path
[list] A list of nodes which defines the path to traverse

Returns

bool
True if `path` is a valid path in `G`

4.6.2 path_weight

path_weight(*G*, *path*, *weight*)

Returns total cost associated with specified path and weight

Parameters

G
[graph] A NetworkX graph.

path: list
A list of node labels which defines the path to traverse

weight: string
A string indicating which edge attribute to use for path cost

Returns

cost: int or float
An integer or a float representing the total cost with respect to the specified weight of the specified path

Raises

NetworkXNoPath
If the specified edge does not exist.

4.7 Freezing graph structure

freeze(G)	Modify graph to prevent further change by adding or removing nodes or edges.
is_frozen(G)	Returns True if graph is frozen.

4.7.1 freeze

freeze(*G*)

Modify graph to prevent further change by adding or removing nodes or edges.

Node and edge data can still be modified.

Parameters

G
[graph] A NetworkX graph

See also:

is_frozen

Notes

To "unfreeze" a graph you must make a copy by creating a new graph object:

```
>>> graph = nx.path_graph(4)
>>> frozen_graph = nx.freeze(graph)
>>> unfrozen_graph = nx.Graph(frozen_graph)
>>> nx.is_frozen(unfrozen_graph)
False
```

Examples

```
>>> G = nx.path_graph(4)
>>> G = nx.freeze(G)
>>> try:
...     G.add_edge(4, 5)
... except nx.NetworkXError as err:
...     print(str(err))
Frozen graph can't be modified
```

4.7.2 is_frozen

is_frozen(*G*)

Returns True if graph is frozen.

Parameters

G

[graph] A NetworkX graph

See also:

freeze

CHAPTER

FIVE

GRAPH GENERATORS

5.1 Atlas

Generators for the small graph atlas.

graph_atlas(i)	Returns graph number `i` from the Graph Atlas.
graph_atlas_g()	Returns the list of all graphs with up to seven nodes named in the Graph Atlas.

5.1.1 graph_atlas

graph_atlas(*i*)

Returns graph number `i` from the Graph Atlas.

For more information, see *graph_atlas_g()*.

Parameters

i

[int] The index of the graph from the atlas to get. The graph at index 0 is assumed to be the null graph.

Returns

list

A list of *Graph* objects, the one at index *i* corresponding to the graph *i* in the Graph Atlas.

See also:

graph_atlas_g

Notes

The time required by this function increases linearly with the argument `i`, since it reads a large file sequentially in order to generate the graph [1].

References

[1]

5.1.2 graph_atlas_g

graph_atlas_g()

Returns the list of all graphs with up to seven nodes named in the Graph Atlas.

The graphs are listed in increasing order by

1. number of nodes,
2. number of edges,
3. degree sequence (for example 111223 < 112222),
4. number of automorphisms,

in that order, with three exceptions as described in the *Notes* section below. This causes the list to correspond with the index of the graphs in the Graph Atlas [atlas], with the first graph, `G[0]`, being the null graph.

Returns

list
A list of *Graph* objects, the one at index *i* corresponding to the graph *i* in the Graph Atlas.

See also:

graph_atlas

Notes

This function may be expensive in both time and space, since it reads a large file sequentially in order to populate the list.

Although the NetworkX atlas functions match the order of graphs given in the "Atlas of Graphs" book, there are (at least) three errors in the ordering described in the book. The following three pairs of nodes violate the lexicographically nondecreasing sorted degree sequence rule:

- graphs 55 and 56 with degree sequences 001111 and 000112,
- graphs 1007 and 1008 with degree sequences 3333444 and 3333336,
- graphs 1012 and 1213 with degree sequences 1244555 and 1244456.

References

[atlas]

5.2 Classic

Generators for some classic graphs.

The typical graph builder function is called as follows:

```
>>> G = nx.complete_graph(100)
```

returning the complete graph on n nodes labeled 0, .., 99 as a simple graph. Except for *empty_graph*, all the functions in this module return a Graph class (i.e. a simple, undirected graph).

balanced_tree(r, h[, create_using])	Returns the perfectly balanced r-ary tree of height h.
barbell_graph(m1, m2[, create_using])	Returns the Barbell Graph: two complete graphs connected by a path.
binomial_tree(n[, create_using])	Returns the Binomial Tree of order n.
complete_graph(n[, create_using])	Return the complete graph K_n with n nodes.
complete_multipartite_graph(*subset_sizes)	Returns the complete multipartite graph with the specified subset sizes.
circular_ladder_graph(n[, create_using])	Returns the circular ladder graph CL_n of length n.
circulant_graph(n, offsets[, create_using])	Returns the circulant graph $Ci_n(x_1, x_2, ..., x_m)$ with n nodes.
cycle_graph(n[, create_using])	Returns the cycle graph C_n of cyclically connected nodes.
dorogovtsev_goltsev_mendes_graph(n[, ...])	Returns the hierarchically constructed Dorogovtsev-Goltsev-Mendes graph.
empty_graph([n, create_using, default])	Returns the empty graph with n nodes and zero edges.
full_rary_tree(r, n[, create_using])	Creates a full r-ary tree of n nodes.
ladder_graph(n[, create_using])	Returns the Ladder graph of length n.
lollipop_graph(m, n[, create_using])	Returns the Lollipop Graph; K_m connected to P_n.
null_graph([create_using])	Returns the Null graph with no nodes or edges.
path_graph(n[, create_using])	Returns the Path graph P_n of linearly connected nodes.
star_graph(n[, create_using])	Return the star graph
trivial_graph([create_using])	Return the Trivial graph with one node (with label 0) and no edges.
turan_graph(n, r)	Return the Turan Graph
wheel_graph(n[, create_using])	Return the wheel graph

5.2.1 balanced_tree

balanced_tree(*r*, *h*, *create_using=None*)

Returns the perfectly balanced r-ary tree of height h.

Parameters

r
[int] Branching factor of the tree; each node will have r children.

h
[int] Height of 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] A balanced `r`-ary tree of height `h`.

Notes

This is the rooted tree where all leaves are at distance `h` from the root. The root has degree `r` and all other internal nodes have degree `r + 1`.

Node labels are integers, starting from zero.

A balanced tree is also known as a *complete r-ary tree*.

5.2.2 barbell_graph

barbell_graph(*m1*, *m2*, *create_using=None*)

Returns the Barbell Graph: two complete graphs connected by a path.

Parameters

m1
[int] Size of the left and right barbells, must be greater than 2.

m2
[int] Length of the path connecting the barbells.

create_using
[NetworkX graph constructor, optional (default=nx.Graph)] Graph type to create. If graph instance, then cleared before populated. Only undirected Graphs are supported.

Returns

G
[NetworkX graph] A barbell graph.

Notes

Two identical complete graphs K_{m1} form the left and right bells, and are connected by a path P_{m2}.

The `2*m1+m2` nodes are numbered
`0, ..., m1-1` for the left barbell, `m1, ..., m1+m2-1` for the path, and `m1+m2, ..., 2*m1+m2-1` for the right barbell.

The 3 subgraphs are joined via the edges `(m1-1, m1)` and `(m1+m2-1, m1+m2)`. If `m2=0`, this is merely two complete graphs joined together.

This graph is an extremal example in David Aldous and Jim Fill's e-text on Random Walks on Graphs.

5.2.3 binomial_tree

binomial_tree(*n*, *create_using=None*)

Returns the Binomial Tree of order n.

The binomial tree of order 0 consists of a single node. A binomial tree of order k is defined recursively by linking two binomial trees of order k-1: the root of one is the leftmost child of the root of the other.

Parameters

n
[int] Order of the binomial 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] A binomial tree of 2^n nodes and $2^n - 1$ edges.

5.2.4 complete_graph

complete_graph(*n*, *create_using=None*)

Return the complete graph `K_n` with n nodes.

A complete graph on `n` nodes means that all pairs of distinct nodes have an edge connecting them.

Parameters

n
[int or iterable container of nodes] If n is an integer, nodes are from range(n). If n is a container of nodes, those nodes appear in the graph. 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.

Examples

```
>>> G = nx.complete_graph(9)
>>> len(G)
9
>>> G.size()
36
>>> G = nx.complete_graph(range(11, 14))
>>> list(G.nodes())
[11, 12, 13]
>>> G = nx.complete_graph(4, nx.DiGraph())
>>> G.is_directed()
True
```

5.2.5 complete_multipartite_graph

complete_multipartite_graph(**subset_sizes*)

Returns the complete multipartite graph with the specified subset sizes.

Parameters

subset_sizes
[tuple of integers or tuple of node iterables] The arguments can either all be integer number of nodes or they can all be iterables of nodes. If integers, they represent the number of nodes in each subset of the multipartite graph. If iterables, each is used to create the nodes for that subset. The length of subset_sizes is the number of subsets.

Returns

G
[NetworkX Graph] Returns the complete multipartite graph with the specified subsets.

For each node, the node attribute 'subset' is an integer indicating which subset contains the node.

See also:

complete_bipartite_graph

Notes

This function generalizes several other graph builder functions.

- If no subset sizes are given, this returns the null graph.
- If a single subset size `n` is given, this returns the empty graph on `n` nodes.
- If two subset sizes `m` and `n` are given, this returns the complete bipartite graph on `m + n` nodes.
- If subset sizes `1` and `n` are given, this returns the star graph on `n + 1` nodes.

Examples

Creating a complete tripartite graph, with subsets of one, two, and three nodes, respectively.

```
>>> G = nx.complete_multipartite_graph(1, 2, 3)
>>> [G.nodes[u]["subset"] for u in G]
[0, 1, 1, 2, 2, 2]
>>> list(G.edges(0))
[(0, 1), (0, 2), (0, 3), (0, 4), (0, 5)]
>>> list(G.edges(2))
[(2, 0), (2, 3), (2, 4), (2, 5)]
>>> list(G.edges(4))
[(4, 0), (4, 1), (4, 2)]
```

```
>>> G = nx.complete_multipartite_graph("a", "bc", "def")
>>> [G.nodes[u]["subset"] for u in sorted(G)]
[0, 1, 1, 2, 2, 2]
```

5.2.6 circular_ladder_graph

circular_ladder_graph(*n*, *create_using=None*)

Returns the circular ladder graph CL_n of length n.

CL_n consists of two concentric n-cycles in which each of the n pairs of concentric nodes are joined by an edge.

Node labels are the integers 0 to n-1

5.2.7 circulant_graph

circulant_graph(*n*, *offsets*, *create_using=None*)

Returns the circulant graph $Ci_n(x_1, x_2, ..., x_m)$ with n nodes.

The circulant graph $Ci_n(x_1, ..., x_m)$ consists of n nodes $0, ..., n-1$ such that node i is connected to nodes $(i+x) \mod n$ and $(i-x) \mod n$ for all x in $x_1, ..., x_m$. Thus $Ci_n(1)$ is a cycle graph.

Parameters

n
[integer] The number of nodes in the graph.

offsets
[list of integers] A list of node offsets, x_1 up to x_m, as described above.

create_using
[NetworkX graph constructor, optional (default=nx.Graph)] Graph type to create. If graph instance, then cleared before populated.

Returns

NetworkX Graph of type create_using

Examples

Many well-known graph families are subfamilies of the circulant graphs; for example, to create the cycle graph on n points, we connect every node to nodes on either side (with offset plus or minus one). For n = 10,

```
>>> G = nx.circulant_graph(10, [1])
>>> edges = [
...     (0, 9),
...     (0, 1),
...     (1, 2),
...     (2, 3),
...     (3, 4),
...     (4, 5),
...     (5, 6),
...     (6, 7),
...     (7, 8),
...     (8, 9),
... ]
...
>>> sorted(edges) == sorted(G.edges())
True
```

Similarly, we can create the complete graph on 5 points with the set of offsets [1, 2]:

```
>>> G = nx.circulant_graph(5, [1, 2])
>>> edges = [
...     (0, 1),
...     (0, 2),
...     (0, 3),
...     (0, 4),
...     (1, 2),
...     (1, 3),
...     (1, 4),
...     (2, 3),
...     (2, 4),
...     (3, 4),
... ]
...
>>> sorted(edges) == sorted(G.edges())
True
```

5.2.8 cycle_graph

cycle_graph(*n*, *create_using=None*)

Returns the cycle graph C_n of cyclically connected nodes.

C_n is a path with its two end-nodes connected.

Parameters

n
: [int or iterable container of nodes] If n is an integer, nodes are from `range(n)`. If n is a container of nodes, those nodes appear in the graph. 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

If create_using is directed, the direction is in increasing order.

5.2.9 dorogovtsev_goltsev_mendes_graph

dorogovtsev_goltsev_mendes_graph(*n*, *create_using=None*)

Returns the hierarchically constructed Dorogovtsev-Goltsev-Mendes graph.

n is the generation. See: arXiv:/cond-mat/0112143 by Dorogovtsev, Goltsev and Mendes.

5.2.10 empty_graph

empty_graph(*n=0*, *create_using=None*, *default=<class 'networkx.classes.graph.Graph'>*)

Returns the empty graph with n nodes and zero edges.

Parameters

n
[int or iterable container of nodes (default = 0)] If n is an integer, nodes are from `range(n)`. If n is a container of nodes, those nodes appear in the graph.

create_using
[Graph Instance, Constructor or None] Indicator of type of graph to return. If a Graph-type instance, then clear and use it. If None, use the `default` constructor. If a constructor, call it to create an empty graph.

default
[Graph constructor (optional, default = nx.Graph)] The constructor to use if create_using is None. If None, then nx.Graph is used. This is used when passing an unknown `create_using` value through your home-grown function to *empty_graph* and you want a default constructor other than nx.Graph.

Notes

The variable create_using should be a Graph Constructor or a "graph"-like object. Constructors, e.g. `nx.Graph` or `nx.MultiGraph` will be used to create the returned graph. "graph"-like objects will be cleared (nodes and edges will be removed) and refitted as an empty "graph" with nodes specified in n. This capability is useful for specifying the class-nature of the resulting empty "graph" (i.e. Graph, DiGraph, MyWeirdGraphClass, etc.).

The variable create_using has three main uses: Firstly, the variable create_using can be used to create an empty digraph, multigraph, etc. For example,

```
>>> n = 10
>>> G = nx.empty_graph(n, create_using=nx.DiGraph)
```

will create an empty digraph on n nodes.

Secondly, one can pass an existing graph (digraph, multigraph, etc.) via create_using. For example, if G is an existing graph (resp. digraph, multigraph, etc.), then empty_graph(n, create_using=G) will empty G (i.e. delete all nodes and edges using G.clear()) and then add n nodes and zero edges, and return the modified graph.

Thirdly, when constructing your home-grown graph creation function you can use empty_graph to construct the graph by passing a user defined create_using to empty_graph. In this case, if you want the default constructor to be other than nx.Graph, specify `default`.

```
>>> def mygraph(n, create_using=None):
...     G = nx.empty_graph(n, create_using, nx.MultiGraph)
...     G.add_edges_from([(0, 1), (0, 1)])
...     return G
>>> G = mygraph(3)
>>> G.is_multigraph()
True
>>> G = mygraph(3, nx.Graph)
>>> G.is_multigraph()
False
```

See also create_empty_copy(G).

Examples

```
>>> G = nx.empty_graph(10)
>>> G.number_of_nodes()
10
>>> G.number_of_edges()
0
>>> G = nx.empty_graph("ABC")
>>> G.number_of_nodes()
3
>>> sorted(G)
['A', 'B', 'C']
```

5.2.11 full_rary_tree

full_rary_tree(*r*, *n*, *create_using=None*)

Creates a full r-ary tree of `n` nodes.

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]

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]

5.2.12 ladder_graph

ladder_graph(*n*, *create_using=None*)

Returns the Ladder graph of length n.

This is two paths of n nodes, with each pair connected by a single edge.

Node labels are the integers 0 to 2*n - 1.

5.2.13 lollipop_graph

lollipop_graph(*m*, *n*, *create_using=None*)

Returns the Lollipop Graph; `K_m` connected to `P_n`.

This is the Barbell Graph without the right barbell.

Parameters

m, n
[int or iterable container of nodes (default = 0)] 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. Make sure you have no duplicates.

The nodes for m appear in the complete graph K_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.

Notes

The 2 subgraphs are joined via an edge (m-1, m). If n=0, this is merely a complete graph.

(This graph is an extremal example in David Aldous and Jim Fill's etext on Random Walks on Graphs.)

5.2.14 null_graph

null_graph(*create_using=None*)

Returns the Null graph with no nodes or edges.

See empty_graph for the use of create_using.

5.2.15 path_graph

path_graph(*n*, *create_using=None*)

Returns the Path graph `P_n` of linearly connected nodes.

Parameters

n
[int or iterable] If an integer, nodes are 0 to n - 1. If an iterable of nodes, in the order they appear in the path. 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.

5.2.16 star_graph

star_graph(*n*, *create_using=None*)

Return the star graph

The star graph consists of one center node connected to n outer nodes.

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)).

5.2.17 trivial_graph

trivial_graph(*create_using=None*)

Return the Trivial graph with one node (with label 0) and no edges.

5.2.18 turan_graph

turan_graph(*n*, *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.

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.

5.2.19 wheel_graph

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.

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.

5.3 Expanders

Provides explicit constructions of expander graphs.

margulis_gabber_galil_graph(n[, create_using])	Returns the Margulis-Gabber-Galil undirected MultiGraph on `n^2` nodes.
chordal_cycle_graph(p[, create_using])	Returns the chordal cycle graph on `p` nodes.
paley_graph(p[, create_using])	Returns the Paley (p-1)/2-regular graph on p nodes.

5.3.1 margulis_gabber_galil_graph

margulis_gabber_galil_graph(*n*, *create_using=None*)

Returns the Margulis-Gabber-Galil undirected MultiGraph on `n^2` nodes.

The undirected MultiGraph is regular with degree `8`. Nodes are integer pairs. The second-largest eigenvalue of the adjacency matrix of the graph is at most `5 sqrt{2}`, regardless of `n`.

Parameters

n
[int] Determines the number of nodes in the graph: `n^2`.

create_using
[NetworkX graph constructor, optional (default MultiGraph)] Graph type to create. If graph instance, then cleared before populated.

Returns

G
[graph] The constructed undirected multigraph.

Raises

NetworkXError
If the graph is directed or not a multigraph.

5.3.2 chordal_cycle_graph

chordal_cycle_graph(*p*, *create_using=None*)

Returns the chordal cycle graph on `p` nodes.

The returned graph is a cycle graph on `p` nodes with chords joining each vertex `x` to its inverse modulo `p`. This graph is a (mildly explicit) 3-regular expander [1].

`p` *must* be a prime number.

Parameters

p
[a prime number] The number of vertices in the graph. This also indicates where the 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]

5.3.3 paley_graph

paley_graph(*p*, *create_using=None*)

Returns the Paley (p-1)/2-regular graph on p nodes.

The returned graph is a graph on Z/pZ with edges between x and y if and only if x-y is a nonzero square in Z/pZ.

If p = 1 mod 4, -1 is a square in Z/pZ 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 = 3 mod 4, -1 is not a square in Z/pZ and therefore either x-y or y-x is a square in Z/pZ 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 Z/pZ. 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).

5.4 Lattice

Functions for generating grid graphs and lattices

The *grid_2d_graph()*, *triangular_lattice_graph()*, and *hexagonal_lattice_graph()* functions correspond to the three regular tilings of the plane, the square, triangular, and hexagonal tilings, respectively. *grid_graph()* and *hypercube_graph()* are similar for arbitrary dimensions. Useful relevant discussion can be found about Triangular Tiling, and Square, Hex and Triangle Grids

grid_2d_graph(m, n[, periodic, create_using])	Returns the two-dimensional grid graph.
grid_graph(dim[, periodic])	Returns the *n*-dimensional grid graph.
hexagonal_lattice_graph(m, n[, periodic, ...])	Returns an `m` by `n` hexagonal lattice graph.
hypercube_graph(n)	Returns the *n*-dimensional hypercube graph.
triangular_lattice_graph(m, n[, periodic, ...])	Returns the m by n triangular lattice graph.

5.4.1 grid_2d_graph

grid_2d_graph(*m*, *n*, *periodic=False*, *create_using=None*)

Returns the two-dimensional grid graph.

The grid graph has each node connected to its four nearest neighbors.

Parameters

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.

5.4.2 grid_graph

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)
6
```

5.4.3 hexagonal_lattice_graph

hexagonal_lattice_graph(*m*, *n*, *periodic=False*, *with_positions=True*, *create_using=None*)

Returns an `m` by `n` hexagonal lattice graph.

The *hexagonal lattice graph* is a graph whose nodes and edges are the hexagonal tiling of the plane.

The returned graph will have `m` rows and `n` columns of hexagons. Odd numbered columns are shifted up relative to even numbered columns.

Positions of nodes are computed by default or `with_positions is True`. Node positions creating the standard embedding in the plane with sidelength 1 and are stored in the node attribute 'pos'. `pos = nx.get_node_attributes(G, 'pos')` creates a dict ready for drawing.

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.

5.4.4 hypercube_graph

hypercube_graph(*n*)

Returns the *n*-dimensional hypercube graph.

The nodes are the integers between 0 and `2 ** n - 1`, inclusive.

For more information on the hypercube graph, see the Wikipedia article Hypercube graph.

Parameters

n
[int] The dimension of the hypercube. The number of nodes in the graph will be `2 ** n`.

Returns

NetworkX graph
The hypercube graph of dimension *n*.

5.4.5 triangular_lattice_graph

triangular_lattice_graph(*m*, *n*, *periodic=False*, *with_positions=True*, *create_using=None*)

Returns the m by n triangular lattice graph.

The triangular lattice graph is a two-dimensional grid graph in which each square unit has a diagonal edge (each grid unit has a chord).

The returned graph has m rows and n columns of triangles. Rows and columns include both triangles pointing up and down. Rows form a strip of constant height. Columns form a series of diamond shapes, staggered with the columns on either side. Another way to state the size is that the nodes form a grid of `m+1` rows and `(n + 1) // 2` columns. The odd row nodes are shifted horizontally relative to the even rows.

Directed graph types have edges pointed up or right.

Positions of nodes are computed by default or `with_positions is True`. The position of each node (embedded in a euclidean plane) is stored in the graph using equilateral triangles with sidelength 1. The height between rows of nodes is thus $\sqrt{(3)}/2$. Nodes lie in the first quadrant with the node $(0, 0)$ at the origin.

Parameters

m
: [int] The number of rows in the lattice.

n
: [int] The number of columns in the lattice.

periodic
: [bool (default: False)] If True, join the boundary vertices of the grid using periodic boundary conditions. The join between boundaries is the final row and column of triangles. This means there is one row and one column fewer nodes for the periodic lattice. Periodic lattices require `m >= 3`, `n >= 5` and are allowed but misaligned if `m` or `n` are odd

with_positions
: [bool (default: True)] Store the coordinates of each node in the graph node attribute 'pos'. The coordinates provide a lattice with equilateral triangles. 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.

Returns

NetworkX graph
: The *m* by *n* triangular lattice graph.

5.5 Small

Various small and named graphs, together with some compact generators.

LCF_graph(n, shift_list, repeats[, create_using])	Return the cubic graph specified in LCF notation.
bull_graph([create_using])	Returns the Bull Graph
chvatal_graph([create_using])	Returns the Chvátal Graph
cubical_graph([create_using])	Returns the 3-regular Platonic Cubical Graph
desargues_graph([create_using])	Returns the Desargues Graph
diamond_graph([create_using])	Returns the Diamond graph
dodecahedral_graph([create_using])	Returns the Platonic Dodecahedral graph.
frucht_graph([create_using])	Returns the Frucht Graph.
heawood_graph([create_using])	Returns the Heawood Graph, a (3,6) cage.
hoffman_singleton_graph()	Returns the Hoffman-Singleton Graph.
house_graph([create_using])	Returns the House graph (square with triangle on top)
house_x_graph([create_using])	Returns the House graph with a cross inside the house square.
icosahedral_graph([create_using])	Returns the Platonic Icosahedral graph.
krackhardt_kite_graph([create_using])	Returns the Krackhardt Kite Social Network.
moebius_kantor_graph([create_using])	Returns the Moebius-Kantor graph.
octahedral_graph([create_using])	Returns the Platonic Octahedral graph.
pappus_graph()	Returns the Pappus graph.
petersen_graph([create_using])	Returns the Petersen graph.
sedgewick_maze_graph([create_using])	Return a small maze with a cycle.
tetrahedral_graph([create_using])	Returns the 3-regular Platonic Tetrahedral graph.
truncated_cube_graph([create_using])	Returns the skeleton of the truncated cube.
truncated_tetrahedron_graph([create_using])	Returns the skeleton of the truncated Platonic tetrahedron.
tutte_graph([create_using])	Returns the Tutte graph.

5.5.1 LCF_graph

LCF_graph(*n*, *shift_list*, *repeats*, *create_using=None*)

Return the cubic graph specified in LCF notation.

LCF notation (LCF=Lederberg-Coxeter-Fruchte) is a compressed notation used in the generation of various cubic Hamiltonian graphs of high symmetry. See, for example, dodecahedral_graph, desargues_graph, heawood_graph and pappus_graph below.

n (number of nodes)
The starting graph is the n-cycle with nodes 0,…,n-1. (The null graph is returned if n < 0.)

shift_list = [s1,s2,..,sk], a list of integer shifts mod n,

repeats
integer specifying the number of times that shifts in shift_list are successively applied to each v_current in the n-cycle to generate an edge between v_current and v_current+shift mod n.

For v1 cycling through the n-cycle a total of k*repeats with shift cycling through shiftlist repeats times connect v1 with v1+shift mod n

The utility graph $K_{3,3}$

```
>>> G = nx.LCF_graph(6, [3, -3], 3)
```

The Heawood graph

```
>>> G = nx.LCF_graph(14, [5, -5], 7)
```

See http://mathworld.wolfram.com/LCFNotation.html for a description and references.

5.5.2 bull_graph

bull_graph(*create_using=None*)

Returns the Bull Graph

The Bull Graph has 5 nodes and 5 edges. It is a planar undirected graph in the form of a triangle with two disjoint pendant edges [1] The name comes from the triangle and pendant edges representing respectively the body and legs of a bull.

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] A bull graph with 5 nodes

References

[1]

5.5.3 chvatal_graph

chvatal_graph(*create_using=None*)

Returns the Chvátal Graph

The Chvátal Graph is an undirected graph with 12 nodes and 24 edges [1]. It has 370 distinct (directed) Hamiltonian cycles, giving a unique generalized LCF notation of order 4, two of order 6 , and 43 of order 1 [2].

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] The Chvátal graph with 12 nodes and 24 edges

References

[1], [2]

5.5.4 cubical_graph

cubical_graph(*create_using=None*)

Returns the 3-regular Platonic Cubical Graph

The skeleton of the cube (the nodes and edges) form a graph, with 8 nodes, and 12 edges. It is a special case of the hypercube graph. It is one of 5 Platonic graphs, each a skeleton of its Platonic solid [1]. Such graphs arise in parallel processing in computers.

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] A cubical graph with 8 nodes and 12 edges

References

[1]

5.5.5 desargues_graph

desargues_graph(*create_using=None*)

Returns the Desargues Graph

The Desargues Graph is a non-planar, distance-transitive cubic graph with 20 nodes and 30 edges [1]. It is a symmetric graph. It can be represented in LCF notation as [5,-5,9,-9]^5 [2].

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] Desargues Graph with 20 nodes and 30 edges

References

[1], [2]

5.5.6 diamond_graph

diamond_graph(*create_using=None*)

Returns the Diamond graph

The Diamond Graph is planar undirected graph with 4 nodes and 5 edges. It is also sometimes known as the double triangle graph or kite 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] Diamond Graph with 4 nodes and 5 edges

References

[1]

5.5.7 dodecahedral_graph

dodecahedral_graph(*create_using=None*)

Returns the Platonic Dodecahedral graph.

The dodecahedral graph has 20 nodes and 30 edges. The skeleton of the dodecahedron forms a graph. It is one of 5 Platonic graphs [1]. It can be described in LCF notation as: `[10, 7, 4, -4, -7, 10, -4, 7, -7, 4]^2` [2].

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] Dodecahedral Graph with 20 nodes and 30 edges

References

[1], [2]

5.5.8 frucht_graph

frucht_graph(*create_using=None*)

Returns the Frucht Graph.

The Frucht Graph is the smallest cubical graph whose automorphism group consists only of the identity element [1]. It has 12 nodes and 18 edges and no nontrivial symmetries. It is planar and Hamiltonian [2].

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] Frucht Graph with 12 nodes and 18 edges

References

[1], [2]

5.5.9 heawood_graph

heawood_graph(*create_using=None*)

Returns the Heawood Graph, a (3,6) cage.

The Heawood Graph is an undirected graph with 14 nodes and 21 edges, named after Percy John Heawood [1]. It is cubic symmetric, nonplanar, Hamiltonian, and can be represented in LCF notation as `[5,-5]^7` [2]. It is the unique (3,6)-cage: the regular cubic graph of girth 6 with minimal number of vertices [3].

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] Heawood Graph with 14 nodes and 21 edges

References

[1], [2], [3]

5.5.10 hoffman_singleton_graph

hoffman_singleton_graph()

Returns the Hoffman-Singleton Graph.

The Hoffman–Singleton graph is a symmetrical undirected graph with 50 nodes and 175 edges. All indices lie in `Z % 5`: that is, the integers mod 5 [1]. It is the only regular graph of vertex degree 7, diameter 2, and girth 5. It is the unique (7,5)-cage graph and Moore graph, and contains many copies of the Petersen graph [2].

Returns

G
[networkx Graph] 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 $h \cdot i + j$ of Q_i [3].

References

[1], [2], [3]

5.5.11 house_graph

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]

5.5.12 house_x_graph

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]

5.5.13 icosahedral_graph

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]

5.5.14 krackhardt_kite_graph

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]

5.5.15 moebius_kantor_graph

moebius_kantor_graph(*create_using=None*)

Returns the Moebius-Kantor graph.

The Möbius-Kantor graph is the cubic symmetric graph on 16 nodes. Its LCF notation is [5,-5]^8, and it is isomorphic to the generalized Petersen 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] Moebius-Kantor graph

References

[1]

5.5.16 octahedral_graph

octahedral_graph(*create_using=None*)

Returns the Platonic Octahedral graph.

The octahedral graph is the 6-node 12-edge Platonic graph having the connectivity of the octahedron [1]. If 6 couples go to a party, and each person shakes hands with every person except his or her partner, then this graph describes the set of handshakes that take place; for this reason it is also called the cocktail party graph [2].

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] Octahedral graph

References

[1], [2]

5.5.17 pappus_graph

pappus_graph()

Returns the Pappus graph.

The Pappus graph is a cubic symmetric distance-regular graph with 18 nodes and 27 edges. It is Hamiltonian and can be represented in LCF notation as [5,7,-7,7,-7,-5]^3 [1].

Returns

G
[networkx Graph] Pappus graph

References

[1]

5.5.18 petersen_graph

petersen_graph(*create_using=None*)

Returns the Petersen graph.

The Peterson graph is a cubic, undirected graph with 10 nodes and 15 edges [1]. Julius Petersen constructed the graph as the smallest counterexample against the claim that a connected bridgeless cubic graph has an edge colouring with three colours [2].

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] Petersen graph

References

[1], [2]

5.5.19 sedgewick_maze_graph

sedgewick_maze_graph(*create_using=None*)

Return a small maze with a cycle.

This is the maze used in Sedgewick, 3rd Edition, Part 5, Graph Algorithms, Chapter 18, e.g. Figure 18.2 and following [1]. Nodes are numbered 0,..,7

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] Small maze with a cycle

References

[1]

5.5.20 tetrahedral_graph

tetrahedral_graph(*create_using=None*)

Returns the 3-regular Platonic Tetrahedral graph.

Tetrahedral graph has 4 nodes and 6 edges. It is a special case of the complete graph, K4, and wheel graph, W4. It is one of the 5 platonic graphs [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] Tetrahedral Grpah

References

[1]

5.5.21 truncated_cube_graph

truncated_cube_graph(*create_using=None*)

Returns the skeleton of the truncated cube.

The truncated cube is an Archimedean solid with 14 regular faces (6 octagonal and 8 triangular), 36 edges and 24 nodes [1]. The truncated cube is created by truncating (cutting off) the tips of the cube one third of the way into each edge [2].

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] Skeleton of the truncated cube

References

[1], [2]

5.5.22 truncated_tetrahedron_graph

truncated_tetrahedron_graph(*create_using=None*)

Returns the skeleton of the truncated Platonic tetrahedron.

The truncated tetrahedron is an Archimedean solid with 4 regular hexagonal faces, 4 equilateral triangle faces, 12 nodes and 18 edges. It can be constructed by truncating all 4 vertices of a regular tetrahedron at one third of the original edge length [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] Skeleton of the truncated tetrahedron

References

[1]

5.5.23 tutte_graph

tutte_graph(*create_using=None*)

Returns the Tutte graph.

The Tutte graph is a cubic polyhedral, non-Hamiltonian graph. It has 46 nodes and 69 edges. It is a counterexample to Tait's conjecture that every 3-regular polyhedron has a Hamiltonian cycle. It can be realized geometrically from a tetrahedron by multiply truncating three of its vertices [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] Tutte graph

References

[1]

5.6 Random Graphs

Generators for random graphs.

fast_gnp_random_graph(n, p[, seed, directed])	Returns a $G_{n,p}$ random graph, also known as an Erdős-Rényi graph or a binomial graph.
gnp_random_graph(n, p[, seed, directed])	Returns a $G_{n,p}$ random graph, also known as an Erdős-Rényi graph or a binomial graph.
dense_gnm_random_graph(n, m[, seed])	Returns a $G_{n,m}$ random graph.
gnm_random_graph(n, m[, seed, directed])	Returns a $G_{n,m}$ random graph.
erdos_renyi_graph(n, p[, seed, directed])	Returns a $G_{n,p}$ random graph, also known as an Erdős-Rényi graph or a binomial graph.
binomial_graph(n, p[, seed, directed])	Returns a $G_{n,p}$ random graph, also known as an Erdős-Rényi graph or a binomial graph.
newman_watts_strogatz_graph(n, k, p[, seed])	Returns a Newman–Watts–Strogatz small-world graph.
watts_strogatz_graph(n, k, p[, seed])	Returns a Watts–Strogatz small-world graph.
connected_watts_strogatz_graph(n, k, p[, ...])	Returns a connected Watts–Strogatz small-world graph.
random_regular_graph(d, n[, seed])	Returns a random d-regular graph on n nodes.
barabasi_albert_graph(n, m[, seed, ...])	Returns a random graph using Barabási–Albert preferential attachment
dual_barabasi_albert_graph(n, m1, m2, p[, ...])	Returns a random graph using dual Barabási–Albert preferential attachment
extended_barabasi_albert_graph(n, m, p, q[, ...])	Returns an extended Barabási–Albert model graph.
powerlaw_cluster_graph(n, m, p[, seed])	Holme and Kim algorithm for growing graphs with powerlaw degree distribution and approximate average clustering.
random_kernel_graph(n, kernel_integral[, ...])	Returns an random graph based on the specified kernel.
random_lobster(n, p1, p2[, seed])	Returns a random lobster graph.
random_shell_graph(constructor[, seed])	Returns a random shell graph for the constructor given.
random_powerlaw_tree(n[, gamma, seed, tries])	Returns a tree with a power law degree distribution.
random_powerlaw_tree_sequence(n[, gamma, ...])	Returns a degree sequence for a tree with a power law distribution.
random_kernel_graph(n, kernel_integral[, ...])	Returns an random graph based on the specified kernel.

5.6.1 fast_gnp_random_graph

fast_gnp_random_graph(*n*, *p*, *seed=None*, *directed=False*)

Returns a $G_{n,p}$ random graph, also known as an Erdős-Rényi graph or a binomial graph.

Parameters

n
[int] The number of nodes.

p
[float] Probability for edge creation.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

directed
[bool, optional (default=False)] If True, this function returns a directed graph.

See also:

gnp_random_graph

Notes

The $G_{n,p}$ graph algorithm chooses each of the $[n(n-1)]/2$ (undirected) or $n(n-1)$ (directed) possible edges with probability p.

This algorithm [1] runs in $O(n+m)$ time, where $\mathtt{m}$ is the expected number of edges, which equals $pn(n-1)/2$. This should be faster than `gnp_random_graph()` when p is small and the expected number of edges is small (that is, the graph is sparse).

References

[1]

5.6.2 gnp_random_graph

gnp_random_graph(*n*, *p*, *seed=None*, *directed=False*)

Returns a $G_{n,p}$ random graph, also known as an Erdős-Rényi graph or a binomial graph.

The $G_{n,p}$ model chooses each of the possible edges with probability p.

Parameters

n
[int] The number of nodes.

p
[float] Probability for edge creation.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

directed
[bool, optional (default=False)] If True, this function returns a directed graph.

See also:

fast_gnp_random_graph

Notes

This algorithm [2] runs in $O(n^2)$ time. For sparse graphs (that is, for small values of p), *fast_gnp_random_graph()* is a faster algorithm.

binomial_graph() and *erdos_renyi_graph()* are aliases for *gnp_random_graph()*.

```
>>> nx.binomial_graph is nx.gnp_random_graph
True
>>> nx.erdos_renyi_graph is nx.gnp_random_graph
True
```

References

[1], [2]

5.6.3 dense_gnm_random_graph

dense_gnm_random_graph(*n*, *m*, *seed=None*)

Returns a $G_{n,m}$ random graph.

In the $G_{n,m}$ model, a graph is chosen uniformly at random from the set of all graphs with n nodes and m edges.

This algorithm should be faster than *gnm_random_graph()* for dense graphs.

Parameters

n
: [int] The number of nodes.

m
: [int] The number of edges.

seed
: [integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

See also:

gnm_random_graph

Notes

Algorithm by Keith M. Briggs Mar 31, 2006. Inspired by Knuth's Algorithm S (Selection sampling technique), in section 3.4.2 of [1].

References

[1]

5.6.4 gnm_random_graph

gnm_random_graph(*n*, *m*, *seed=None*, *directed=False*)

Returns a $G_{n,m}$ random graph.

In the $G_{n,m}$ model, a graph is chosen uniformly at random from the set of all graphs with n nodes and m edges.

This algorithm should be faster than `dense_gnm_random_graph()` for sparse graphs.

Parameters

n
[int] The number of nodes.

m
[int] The number of edges.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

directed
[bool, optional (default=False)] If True return a directed graph

See also:

dense_gnm_random_graph

5.6.5 erdos_renyi_graph

erdos_renyi_graph(*n*, *p*, *seed=None*, *directed=False*)

Returns a $G_{n,p}$ random graph, also known as an Erdős-Rényi graph or a binomial graph.

The $G_{n,p}$ model chooses each of the possible edges with probability p.

Parameters

n
[int] The number of nodes.

p
[float] Probability for edge creation.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

directed
[bool, optional (default=False)] If True, this function returns a directed graph.

See also:

fast_gnp_random_graph

Notes

This algorithm [2] runs in $O(n^2)$ time. For sparse graphs (that is, for small values of p), *fast_gnp_random_graph()* is a faster algorithm.

binomial_graph() and *erdos_renyi_graph()* are aliases for *gnp_random_graph()*.

```
>>> nx.binomial_graph is nx.gnp_random_graph
True
>>> nx.erdos_renyi_graph is nx.gnp_random_graph
True
```

References

[1], [2]

5.6.6 binomial_graph

binomial_graph(*n*, *p*, *seed=None*, *directed=False*)

Returns a $G_{n,p}$ random graph, also known as an Erdős-Rényi graph or a binomial graph.

The $G_{n,p}$ model chooses each of the possible edges with probability p.

Parameters

n
[int] The number of nodes.

p
[float] Probability for edge creation.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

directed
[bool, optional (default=False)] If True, this function returns a directed graph.

See also:

fast_gnp_random_graph

Notes

This algorithm [2] runs in $O(n^2)$ time. For sparse graphs (that is, for small values of p), *fast_gnp_random_graph()* is a faster algorithm.

binomial_graph() and *erdos_renyi_graph()* are aliases for *gnp_random_graph()*.

```
>>> nx.binomial_graph is nx.gnp_random_graph
True
>>> nx.erdos_renyi_graph is nx.gnp_random_graph
True
```

References

[1], [2]

5.6.7 newman_watts_strogatz_graph

newman_watts_strogatz_graph(*n*, *k*, *p*, *seed=None*)

Returns a Newman–Watts–Strogatz small-world graph.

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 adding a new edge for each edge.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

See also:

watts_strogatz_graph

Notes

First create a ring over n nodes [1]. Then each node in the ring is connected with its k nearest neighbors (or $k - 1$ neighbors if k is odd). Then shortcuts are created by adding new edges as follows: for each edge (u, v) in the underlying "n-ring with k nearest neighbors" with probability p add a new edge (u, w) with randomly-chosen existing node w. In contrast with `watts_strogatz_graph()`, no edges are removed.

References

[1]

5.6.8 watts_strogatz_graph

watts_strogatz_graph(*n*, *k*, *p*, *seed=None*)

Returns a Watts–Strogatz small-world graph.

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

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

See also:

newman_watts_strogatz_graph
connected_watts_strogatz_graph

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.

In contrast with `newman_watts_strogatz_graph()`, the random rewiring does not increase the number of edges. The rewired graph is not guaranteed to be connected as in `connected_watts_strogatz_graph()`.

References

[1]

5.6.9 connected_watts_strogatz_graph

connected_watts_strogatz_graph(*n*, *k*, *p*, *tries=100*, *seed=None*)

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 *Randomness*.

See also:

newman_watts_strogatz_graph
watts_strogatz_graph

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.

References

[1]

5.6.10 random_regular_graph

random_regular_graph(*d*, *n*, *seed=None*)

Returns a random d-regular graph on n nodes.

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 *Randomness*.

Raises

NetworkXError
If $n \times d$ is odd or d is greater than or equal to n.

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 $d = O(n^{1/3-\epsilon})$.

References

[1], [2]

5.6.11 barabasi_albert_graph

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 *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]

5.6.12 dual_barabasi_albert_graph

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] Number of nodes

m1
[int] Number of edges to link each new node to existing nodes with probability p

m2
[int] Number of edges to link each new node to existing nodes with probability $1-p$

p
[float] The probability of attaching m_1 edges (as opposed to m_2 edges)

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

initial_graph
[Graph or None (default)] Initial network for Barabási–Albert algorithm. A copy of `initial_graph` is used. It should be connected for most use cases. If None, starts from an star graph on max(m1, m2) + 1 nodes.

Returns

G
[Graph]

Raises

NetworkXError
If `m1` and `m2` do not satisfy `1 <= m1,m2 < n`, or `p` does not satisfy `0 <= p <= 1`, or the initial graph number of nodes m0 does not satisfy m1, m2 <= m0 <= n.

References

[1]

5.6.13 extended_barabasi_albert_graph

extended_barabasi_albert_graph(*n*, *m*, *p*, *q*, *seed=None*)

Returns an extended Barabási–Albert model graph.

An extended Barabási–Albert model graph is a random graph constructed using preferential attachment. The extended model allows new edges, rewired edges or new nodes. Based on the probabilities p and q with $p + q < 1$, the growing behavior of the graph is determined as:

1) With p probability, m new edges are added to the graph, starting from randomly chosen existing nodes and attached preferentially at the other end.

2) With q probability, m existing edges are rewired by randomly choosing an edge and rewiring one end to a preferentially chosen node.

3) With $(1 - p - q)$ probability, m new nodes are added to the graph with edges attached preferentially.

When $p = q = 0$, the model behaves just like the Barabási–Alber model.

Parameters

n
[int] Number of nodes

m
[int] Number of edges with which a new node attaches to existing nodes

p
[float] Probability value for adding an edge between existing nodes. p + q < 1

q
[float] Probability value of rewiring of existing edges. p + q < 1

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

G
[Graph]

Raises

NetworkXError
If `m` does not satisfy `1 <= m < n` or `1 >= p + q`

References

[1]

5.6.14 powerlaw_cluster_graph

powerlaw_cluster_graph(*n*, *m*, *p*, *seed=None*)

Holme and Kim algorithm for growing graphs with powerlaw degree distribution and approximate average clustering.

Parameters

n
[int] the number of nodes

m
[int] the number of random edges to add for each new node

p
[float,] Probability of adding a triangle after adding a random edge

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Raises

NetworkXError
If `m` does not satisfy `1 <= m <= n` or `p` does not satisfy `0 <= p <= 1`.

Notes

The average clustering has a hard time getting above a certain cutoff that depends on `m`. This cutoff is often quite low. The transitivity (fraction of triangles to possible triangles) seems to decrease with network size.

It is essentially the Barabási–Albert (BA) growth model with an extra step that each random edge is followed by a chance of making an edge to one of its neighbors too (and thus a triangle).

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.

References

[1]

5.6.15 random_kernel_graph

random_kernel_graph(*n*, *kernel_integral*, *kernel_root=None*, *seed=None*)

Returns an random graph based on the specified kernel.

The algorithm chooses each of the $[n(n-1)]/2$ possible edges with probability specified by a kernel $\kappa(x, y)$ [1]. The kernel $\kappa(x, y)$ must be a symmetric (in x, y), non-negative, bounded function.

Parameters

n
[int] The number of nodes

kernel_integral
[function] Function that returns the definite integral of the kernel $\kappa(x, y)$, $F(y, a, b) := \int_a^b \kappa(x, y)dx$

kernel_root: function (optional)
Function that returns the root b of the equation $F(y, a, b) = r$. If None, the root is found using `scipy.optimize.brentq()` (this requires SciPy).

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

See also:

gnp_random_graph
expected_degree_graph

Notes

The kernel is specified through its definite integral which must be provided as one of the arguments. If the integral and root of the kernel integral can be found in $O(1)$ time then this algorithm runs in time $O(n + m)$ where m is the expected number of edges [2].

The nodes are set to integers from 0 to $n - 1$.

References

[1], [2]

Examples

Generate an Erdős–Rényi random graph $G(n, c/n)$, with kernel $\kappa(x, y) = c$ where c is the mean expected degree.

```
>>> def integral(u, w, z):
...     return c * (z - w)
>>> def root(u, w, r):
...     return r / c + w
>>> c = 1
>>> graph = nx.random_kernel_graph(1000, integral, root)
```

5.6.16 random_lobster

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] Probability of adding an edge one level beyond backbone

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Raises

NetworkXError
If `p1` or `p2` parameters are >= 1 because the while loops would never finish.

5.6.17 random_shell_graph

random_shell_graph(*constructor*, *seed=None*)

Returns a random shell graph for the constructor given.

Parameters

constructor
[list of three-tuples] Represents the parameters for a shell, starting at the center shell. Each element of the list must be of the form `(n, m, d)`, where `n` is the number of nodes in the shell, `m` is the number of edges in the shell, and `d` is the ratio of inter-shell (next) edges to intra-shell edges. If `d` is zero, there will be no intra-shell edges, and if `d` is one there will be all possible intra-shell edges.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Examples

```
>>> constructor = [(10, 20, 0.8), (20, 40, 0.8)]
>>> G = nx.random_shell_graph(constructor)
```

5.6.18 random_powerlaw_tree

random_powerlaw_tree(*n*, *gamma=3*, *seed=None*, *tries=100*)

Returns a tree with a power law degree distribution.

Parameters

n
[int] The number of nodes.

gamma
[float] Exponent of the power law.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

tries
[int] Number of attempts to adjust the sequence to make it a tree.

Raises

NetworkXError
If no valid sequence is found within the maximum number of attempts.

Notes

A trial power law degree sequence is chosen and then elements are swapped with new elements from a powerlaw distribution until the sequence makes a tree (by checking, for example, that the number of edges is one smaller than the number of nodes).

5.6.19 random_powerlaw_tree_sequence

random_powerlaw_tree_sequence(*n*, *gamma=3*, *seed=None*, *tries=100*)

Returns a degree sequence for a tree with a power law distribution.

Parameters

n
[int,] The number of nodes.

gamma
[float] Exponent of the power law.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

tries
[int] Number of attempts to adjust the sequence to make it a tree.

Raises

NetworkXError
If no valid sequence is found within the maximum number of attempts.

Notes

A trial power law degree sequence is chosen and then elements are swapped with new elements from a power law distribution until the sequence makes a tree (by checking, for example, that the number of edges is one smaller than the number of nodes).

5.7 Duplication Divergence

Functions for generating graphs based on the "duplication" method.

These graph generators start with a small initial graph then duplicate nodes and (partially) duplicate their edges. These functions are generally inspired by biological networks.

duplication_divergence_graph(n, p[, seed])	Returns an undirected graph using the duplication-divergence model.
partial_duplication_graph(N, n, p, q[, seed])	Returns a random graph using the partial duplication model.

5.7.1 duplication_divergence_graph

duplication_divergence_graph(*n*, *p*, *seed=None*)

Returns an undirected graph using the duplication-divergence model.

A graph of `n` nodes is created by duplicating the initial nodes and retaining edges incident to the original nodes with a retention probability `p`.

Parameters

n
[int] The desired number of nodes in the graph.

p
[float] The probability for retaining the edge of the replicated node.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

G
[Graph]

Raises

NetworkXError
If `p` is not a valid probability. If `n` is less than 2.

Notes

This algorithm appears in [1].

This implementation disallows the possibility of generating disconnected graphs.

References

[1]

5.7.2 partial_duplication_graph

partial_duplication_graph(*N*, *n*, *p*, *q*, *seed=None*)

Returns a random graph using the partial duplication model.

Parameters

N
[int] The total number of nodes in the final graph.

n
[int] The number of nodes in the initial clique.

p
[float] The probability of joining each neighbor of a node to the duplicate node. Must be a number in the between zero and one, inclusive.

q
[float] The probability of joining the source node to the duplicate node. Must be a number in the between zero and one, inclusive.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Notes

A graph of nodes is grown by creating a fully connected graph of size `n`. The following procedure is then repeated until a total of `N` nodes have been reached.

1. A random node, *u*, is picked and a new node, *v*, is created.
2. For each neighbor of *u* an edge from the neighbor to *v* is created with probability `p`.
3. An edge from *u* to *v* is created with probability `q`.

This algorithm appears in [1].

This implementation allows the possibility of generating disconnected graphs.

References

[1]

5.8 Degree Sequence

Generate graphs with a given degree sequence or expected degree sequence.

configuration_model(deg_sequence[, ...])	Returns a random graph with the given degree sequence.
directed_configuration_model(...[, ...])	Returns a directed_random graph with the given degree sequences.
expected_degree_graph(w[, seed, selfloops])	Returns a random graph with given expected degrees.
havel_hakimi_graph(deg_sequence[, create_using])	Returns a simple graph with given degree sequence constructed using the Havel-Hakimi algorithm.
directed_havel_hakimi_graph(in_deg_sequence, ...)	Returns a directed graph with the given degree sequences.
degree_sequence_tree(deg_sequence[, ...])	Make a tree for the given degree sequence.
random_degree_sequence_graph(sequence[, ...])	Returns a simple random graph with the given degree sequence.

5.8.1 configuration_model

configuration_model(*deg_sequence*, *create_using=None*, *seed=None*)

Returns a random graph with the given degree sequence.

The configuration model generates a random pseudograph (graph with parallel edges and self loops) by randomly assigning edges to match the given degree sequence.

Parameters

deg_sequence
[list of nonnegative integers] Each list entry corresponds to the degree of a node.

create_using
[NetworkX graph constructor, optional (default MultiGraph)] Graph type to create. If graph instance, then cleared before populated.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

G
[MultiGraph] A graph with the specified degree sequence. Nodes are labeled starting at 0 with an index corresponding to the position in deg_sequence.

Raises

NetworkXError
If the degree sequence does not have an even sum.

See also:

is_graphical

Notes

As described by Newman [1].

A non-graphical degree sequence (not realizable by some simple graph) is allowed since this function returns graphs with self loops and parallel edges. An exception is raised if the degree sequence does not have an even sum.

This configuration model construction process can lead to duplicate edges and loops. You can remove the self-loops and parallel edges (see below) which will likely result in a graph that doesn't have the exact degree sequence specified.

The density of self-loops and parallel edges tends to decrease as the number of nodes increases. However, typically the number of self-loops will approach a Poisson distribution with a nonzero mean, and similarly for the number of parallel edges. Consider a node with *k* stubs. The probability of being joined to another stub of the same node is basically (*k* - *1*) / *N*, where *k* is the degree and *N* is the number of nodes. So the probability of a self-loop scales like *c* / *N* for some constant *c*. As *N* grows, this means we expect *c* self-loops. Similarly for parallel edges.

References

[1]

Examples

You can create a degree sequence following a particular distribution by using the one of the distribution functions in *random_sequence* (or one of your own). For example, to create an undirected multigraph on one hundred nodes with degree sequence chosen from the power law distribution:

```
>>> sequence = nx.random_powerlaw_tree_sequence(100, tries=5000)
>>> G = nx.configuration_model(sequence)
>>> len(G)
100
>>> actual_degrees = [d for v, d in G.degree()]
>>> actual_degrees == sequence
True
```

The returned graph is a multigraph, which may have parallel edges. To remove any parallel edges from the returned graph:

```
>>> G = nx.Graph(G)
```

Similarly, to remove self-loops:

```
>>> G.remove_edges_from(nx.selfloop_edges(G))
```

5.8.2 directed_configuration_model

directed_configuration_model(*in_degree_sequence*, *out_degree_sequence*, *create_using=None*, *seed=None*)

Returns a directed_random graph with the given degree sequences.

The configuration model generates a random directed pseudograph (graph with parallel edges and self loops) by randomly assigning edges to match the given degree sequences.

Parameters

in_degree_sequence
[list of nonnegative integers] Each list entry corresponds to the in-degree of a node.

out_degree_sequence
[list of nonnegative integers] Each list entry corresponds to the out-degree of a node.

create_using
[NetworkX graph constructor, optional (default MultiDiGraph)] Graph type to create. If graph instance, then cleared before populated.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

G
[MultiDiGraph] 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 do not have the same sum.

See also:

configuration_model

Notes

Algorithm as described by Newman [1].

A non-graphical degree sequence (not realizable by some simple graph) is allowed since this function returns graphs with self loops and parallel edges. An exception is raised if the degree sequences does not have the same sum.

This configuration model construction process can lead to duplicate edges and loops. You can remove the self-loops and parallel edges (see below) which will likely result in a graph that doesn't have the exact degree sequence specified. This "finite-size effect" decreases as the size of the graph increases.

References

[1]

Examples

One can modify the in- and out-degree sequences from an existing directed graph in order to create a new directed graph. For example, here we modify the directed path graph:

```
>>> D = nx.DiGraph([(0, 1), (1, 2), (2, 3)])
>>> din = list(d for n, d in D.in_degree())
>>> dout = list(d for n, d in D.out_degree())
>>> din.append(1)
>>> dout[0] = 2
>>> # We now expect an edge from node 0 to a new node, node 3.
... D = nx.directed_configuration_model(din, dout)
```

The returned graph is a directed multigraph, which may have parallel edges. To remove any parallel edges from the returned graph:

```
>>> D = nx.DiGraph(D)
```

Similarly, to remove self-loops:

```
>>> D.remove_edges_from(nx.selfloop_edges(D))
```

5.8.3 expected_degree_graph

expected_degree_graph(*w*, *seed=None*, *selfloops=True*)

Returns a random graph with given expected degrees.

Given a sequence of expected degrees $W = (w_0, w_1, \ldots, w_{n-1})$ of length n this algorithm assigns an edge between node u and node v with probability

$$p_{uv} = \frac{w_u w_v}{\sum_k w_k}.$$

Parameters

w
[list] The list of expected degrees.

selfloops: bool (default=True)
Set to False to remove the possibility of self-loop edges.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

Graph

Notes

The nodes have integer labels corresponding to index of expected degrees input sequence.

The complexity of this algorithm is $\mathcal{O}(n + m)$ where n is the number of nodes and m is the expected number of edges.

The model in [1] includes the possibility of self-loop edges. Set selfloops=False to produce a graph without self loops.

For finite graphs this model doesn't produce exactly the given expected degree sequence. Instead the expected degrees are as follows.

For the case without self loops (selfloops=False),

$$E[deg(u)] = \sum_{v \neq u} p_{uv} = w_u \left(1 - \frac{w_u}{\sum_k w_k}\right).$$

NetworkX uses the standard convention that a self-loop edge counts 2 in the degree of a node, so with self loops (selfloops=True),

$$E[deg(u)] = \sum_{v \neq u} p_{uv} + 2p_{uu} = w_u \left(1 + \frac{w_u}{\sum_k w_k}\right).$$

References

[1], [2]

Examples

```
>>> z = [10 for i in range(100)]
>>> G = nx.expected_degree_graph(z)
```

5.8.4 havel_hakimi_graph

havel_hakimi_graph(*deg_sequence*, *create_using=None*)

Returns a simple graph with given degree sequence constructed using the Havel-Hakimi algorithm.

Parameters

deg_sequence: list of integers
Each integer corresponds to the degree of a node (need not be sorted).

create_using
[NetworkX graph constructor, optional (default=nx.Graph)] Graph type to create. If graph instance, then cleared before populated. Directed graphs are not allowed.

Raises

NetworkXException
For a non-graphical degree sequence (i.e. one not realizable by some simple graph).

Notes

The Havel-Hakimi algorithm constructs a simple graph by successively connecting the node of highest degree to other nodes of highest degree, resorting remaining nodes by degree, and repeating the process. The resulting graph has a high degree-associativity. Nodes are labeled 1,.., len(deg_sequence), corresponding to their position in deg_sequence.

The basic algorithm is from Hakimi [1] and was generalized by Kleitman and Wang [2].

References

[1], [2]

5.8.5 directed_havel_hakimi_graph

directed_havel_hakimi_graph(*in_deg_sequence*, *out_deg_sequence*, *create_using=None*)

Returns a directed graph with the given degree sequences.

Parameters

in_deg_sequence
[list of integers] Each list entry corresponds to the in-degree of a node.

out_deg_sequence
[list of integers] Each list entry corresponds to the out-degree of a node.

create_using
[NetworkX graph constructor, optional (default DiGraph)] Graph type to create. If graph instance, then cleared before populated.

Returns

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]

5.8.6 degree_sequence_tree

degree_sequence_tree(*deg_sequence*, *create_using=None*)

Make a tree for the given degree sequence.

A tree has #nodes-#edges=1 so the degree sequence must have len(deg_sequence)-sum(deg_sequence)/2=1

5.8.7 random_degree_sequence_graph

random_degree_sequence_graph(*sequence*, *seed=None*, *tries=10*)

Returns a simple random graph with the given degree sequence.

If the maximum degree d_m in the sequence is $O(m^{1/4})$ then the algorithm produces almost uniform random graphs in $O(md_m)$ time where m is the number of edges.

Parameters

sequence
[list of integers] Sequence of degrees

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

tries
[int, optional] Maximum number of tries to create a graph

Returns

G
[Graph] A graph with the specified degree sequence. Nodes are labeled starting at 0 with an index corresponding to the position in the sequence.

Raises

NetworkXUnfeasible
If the degree sequence is not graphical.

NetworkXError
If a graph is not produced in specified number of tries

See also:

is_graphical, ***configuration_model***

Notes

The generator algorithm [1] is not guaranteed to produce a graph.

References

[1]

Examples

```
>>> sequence = [1, 2, 2, 3]
>>> G = nx.random_degree_sequence_graph(sequence, seed=42)
>>> sorted(d for n, d in G.degree())
[1, 2, 2, 3]
```

5.9 Random Clustered

Generate graphs with given degree and triangle sequence.

random_clustered_graph(joint_degree_sequence)	Generate a random graph with the given joint independent edge degree and triangle degree sequence.

5.9.1 random_clustered_graph

random_clustered_graph(*joint_degree_sequence*, *create_using=None*, *seed=None*)

Generate a random graph with the given joint independent edge degree and triangle degree sequence.

This uses a configuration model-like approach to generate a random graph (with parallel edges and self-loops) by randomly assigning edges to match the given joint degree sequence.

The joint degree sequence is a list of pairs of integers of the form $[(d_{1,i}, d_{1,t}), \ldots, (d_{n,i}, d_{n,t})]$. According to this list, vertex u is a member of $d_{u,t}$ triangles and has $d_{u,i}$ other edges. The number $d_{u,t}$ is the *triangle degree* of u and the number $d_{u,i}$ is the *independent edge degree*.

Parameters

joint_degree_sequence
: [list of integer pairs] Each list entry corresponds to the independent edge degree and triangle degree of a node.

create_using
: [NetworkX graph constructor, optional (default MultiGraph)] Graph type to create. If graph instance, then cleared before populated.

seed
: [integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

G
: [MultiGraph] A graph with the specified degree sequence. Nodes are labeled starting at 0 with an index corresponding to the position in deg_sequence.

Raises

NetworkXError
: If the independent edge degree sequence sum is not even or the triangle degree sequence sum is not divisible by 3.

Notes

As described by Miller [1] (see also Newman [2] for an equivalent description).

A non-graphical degree sequence (not realizable by some simple graph) is allowed since this function returns graphs with self loops and parallel edges. An exception is raised if the independent degree sequence does not have an even sum or the triangle degree sequence sum is not divisible by 3.

This configuration model-like construction process can lead to duplicate edges and loops. You can remove the self-loops and parallel edges (see below) which will likely result in a graph that doesn't have the exact degree sequence specified. This "finite-size effect" decreases as the size of the graph increases.

References

[1], [2]

Examples

```
>>> deg = [(1, 0), (1, 0), (1, 0), (2, 0), (1, 0), (2, 1), (0, 1), (0, 1)]
>>> G = nx.random_clustered_graph(deg)
```

To remove parallel edges:

```
>>> G = nx.Graph(G)
```

To remove self loops:

```
>>> G.remove_edges_from(nx.selfloop_edges(G))
```

5.10 Directed

Generators for some directed graphs, including growing network (GN) graphs and scale-free graphs.

gn_graph(n[, kernel, create_using, seed])	Returns the growing network (GN) digraph with n nodes.
gnr_graph(n, p[, create_using, seed])	Returns the growing network with redirection (GNR) digraph with n nodes and redirection probability p.
gnc_graph(n[, create_using, seed])	Returns the growing network with copying (GNC) digraph with n nodes.
random_k_out_graph(n, k, alpha[, ...])	Returns a random k-out graph with preferential attachment.
scale_free_graph(n[, alpha, beta, gamma, ...])	Returns a scale-free directed graph.

5.10.1 gn_graph

gn_graph(*n*, *kernel=None*, *create_using=None*, *seed=None*)

Returns the growing network (GN) digraph with n nodes.

The GN graph is built by adding nodes one at a time with a link to one previously added node. The target node for the link is chosen with probability based on degree. The default attachment kernel is a linear function of the degree of a node.

The graph is always a (directed) tree.

Parameters

n
[int] The number of nodes for the generated graph.

kernel
[function] The attachment kernel.

create_using
[NetworkX graph constructor, optional (default DiGraph)] Graph type to create. If graph instance, then cleared before populated.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

References

[1]

Examples

To create the undirected GN graph, use the `to_directed()` method:

```
>>> D = nx.gn_graph(10)  # the GN graph
>>> G = D.to_undirected()  # the undirected version
```

To specify an attachment kernel, use the `kernel` keyword argument:

```
>>> D = nx.gn_graph(10, kernel=lambda x: x ** 1.5)  # A_k = k^1.5
```

5.10.2 gnr_graph

gnr_graph(*n*, *p*, *create_using=None*, *seed=None*)

Returns the growing network with redirection (GNR) digraph with `n` nodes and redirection probability `p`.

The GNR graph is built by adding nodes one at a time with a link to one previously added node. The previous target node is chosen uniformly at random. With probabiliy `p` the link is instead "redirected" to the successor node of the target.

The graph is always a (directed) tree.

Parameters

n
: [int] The number of nodes for the generated graph.

p
: [float] The redirection probability.

create_using
: [NetworkX graph constructor, optional (default DiGraph)] Graph type to create. If graph instance, then cleared before populated.

seed
: [integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

References

[1]

Examples

To create the undirected GNR graph, use the `to_directed()` method:

```
>>> D = nx.gnr_graph(10, 0.5)  # the GNR graph
>>> G = D.to_undirected()  # the undirected version
```

5.10.3 gnc_graph

gnc_graph(*n*, *create_using=None*, *seed=None*)

Returns the growing network with copying (GNC) digraph with `n` nodes.

The GNC graph is built by adding nodes one at a time with a link to one previously added node (chosen uniformly at random) and to all of that node's successors.

Parameters

n
[int] The number of nodes for the generated graph.

create_using
[NetworkX graph constructor, optional (default DiGraph)] Graph type to create. If graph instance, then cleared before populated.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

References

[1]

5.10.4 random_k_out_graph

random_k_out_graph(*n*, *k*, *alpha*, *self_loops=True*, *seed=None*)

Returns a random `k`-out graph with preferential attachment.

A random `k`-out graph with preferential attachment is a multidigraph generated by the following algorithm.

1. Begin with an empty digraph, and initially set each node to have weight `alpha`.
2. Choose a node `u` with out-degree less than `k` uniformly at random.
3. Choose a node `v` from with probability proportional to its weight.
4. Add a directed edge from `u` to `v`, and increase the weight of `v` by one.
5. If each node has out-degree `k`, halt, otherwise repeat from step 2.

For more information on this model of random graph, see [1].

Parameters

n
[int] The number of nodes in the returned graph.

k
[int] The out-degree of each node in the returned graph.

alpha
[float] A positive `float` representing the initial weight of each vertex. A higher number means that in step 3 above, nodes will be chosen more like a true uniformly random sample, and a lower number means that nodes are more likely to be chosen as their in-degree increases. If this parameter is not positive, a `ValueError` is raised.

self_loops
[bool] If True, self-loops are allowed when generating the graph.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

MultiDiGraph
A k-out-regular multidigraph generated according to the above algorithm.

Raises

ValueError
If alpha is not positive.

Notes

The returned multidigraph may not be strongly connected, or even weakly connected.

References

[1]: Peterson, Nicholas R., and Boris Pittel.
"Distance between two random k-out digraphs, with and without preferential attachment." arXiv preprint arXiv:1311.5961 (2013). <https://arxiv.org/abs/1311.5961>

5.10.5 scale_free_graph

scale_free_graph(*n*, *alpha=0.41*, *beta=0.54*, *gamma=0.05*, *delta_in=0.2*, *delta_out=0*, *create_using=None*, *seed=None*, *initial_graph=None*)

Returns a scale-free directed graph.

Parameters

n
[integer] Number of nodes in graph

alpha
[float] Probability for adding a new node connected to an existing node chosen randomly according to the in-degree distribution.

beta
[float] Probability for adding an edge between two existing nodes. One existing node is chosen randomly according the in-degree distribution and the other chosen randomly according to the out-degree distribution.

gamma
[float] Probability for adding a new node connected to an existing node chosen randomly according to the out-degree distribution.

delta_in
[float] Bias for choosing nodes from in-degree distribution.

delta_out
[float] Bias for choosing nodes from out-degree distribution.

create_using
[NetworkX graph constructor, optional] The default is a MultiDiGraph 3-cycle. If a graph

instance, use it without clearing first. If a graph constructor, call it to construct an empty graph.

Deprecated since version 3.0: create_using is deprecated, use `initial_graph` instead.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

initial_graph
[MultiDiGraph instance, optional] Build the scale-free graph starting from this initial MultiDiGraph, if provided.

Returns

MultiDiGraph

Notes

The sum of `alpha`, `beta`, and `gamma` must be 1.

References

[1]

Examples

Create a scale-free graph on one hundred nodes:

```
>>> G = nx.scale_free_graph(100)
```

5.11 Geometric

Generators for geometric graphs.

geometric_edges(G, radius[, p])	Returns edge list of node pairs within `radius` of each other.
geographical_threshold_graph(n, theta[, ...])	Returns a geographical threshold graph.
navigable_small_world_graph(n[, p, q, r, ...])	Returns a navigable small-world graph.
random_geometric_graph(n, radius[, dim, ...])	Returns a random geometric graph in the unit cube of dimensions `dim`.
soft_random_geometric_graph(n, radius[, ...])	Returns a soft random geometric graph in the unit cube.
thresholded_random_geometric_graph(n, ...[, ...])	Returns a thresholded random geometric graph in the unit cube.
waxman_graph(n[, beta, alpha, L, domain, ...])	Returns a Waxman random graph.

5.11.1 geometric_edges

geometric_edges(*G*, *radius*, *p=2*)

Returns edge list of node pairs within `radius` of each other.

Parameters

G
[networkx graph] The graph from which to generate the edge list. The nodes in `G` should have an attribute `pos` corresponding to the node position, which is used to compute the distance to other nodes.

radius
[scalar] The distance threshold. Edges are included in the edge list if the distance between the two nodes is less than `radius`.

p
[scalar, default=2] The Minkowski distance metric used to compute distances. The default value is 2, i.e. Euclidean distance.

Returns

edges
[list] List of edges whose distances are less than `radius`

Notes

Radius uses Minkowski distance metric `p`. If scipy is available, `scipy.spatial.cKDTree` is used to speed computation.

Examples

Create a graph with nodes that have a "pos" attribute representing 2D coordinates.

```
>>> G = nx.Graph()
>>> G.add_nodes_from([
...     (0, {"pos": (0, 0)}),
...     (1, {"pos": (3, 0)}),
...     (2, {"pos": (8, 0)}),
... ])
>>> nx.geometric_edges(G, radius=1)
[]
>>> nx.geometric_edges(G, radius=4)
[(0, 1)]
>>> nx.geometric_edges(G, radius=6)
[(0, 1), (1, 2)]
>>> nx.geometric_edges(G, radius=9)
[(0, 1), (0, 2), (1, 2)]
```

5.11.2 geographical_threshold_graph

geographical_threshold_graph(*n*, *theta*, *dim=2*, *pos=None*, *weight=None*, *metric=None*, *p_dist=None*, *seed=None*)

Returns a geographical threshold graph.

The geographical threshold graph model places n nodes uniformly at random in a rectangular domain. Each node u is assigned a weight w_u. Two nodes u and v are joined by an edge if

$$(w_u + w_v)p_{dist}(r) \geq \theta$$

where `r` is the distance between `u` and `v`, `p_dist` is any function of `r`, and θ as the threshold parameter. `p_dist` is used to give weight to the distance between nodes when deciding whether or not they should be connected. The larger `p_dist` is, the more prone nodes separated by `r` are to be connected, and vice versa.

Parameters

n
[int or iterable] Number of nodes or iterable of nodes

theta: float
Threshold value

dim
[int, optional] Dimension of graph

pos
[dict] Node positions as a dictionary of tuples keyed by node.

weight
[dict] Node weights as a dictionary of numbers keyed by node.

metric
[function] A metric on vectors of numbers (represented as lists or tuples). This must be a function that accepts two lists (or tuples) as input and yields a number as output. The function must also satisfy the four requirements of a metric. Specifically, if d is the function and x, y, and z are vectors in the graph, then d must satisfy

1. $d(x, y) \geq 0$,
2. $d(x, y) = 0$ if and only if $x = y$,
3. $d(x, y) = d(y, x)$,
4. $d(x, z) \leq d(x, y) + d(y, z)$.

If this argument is not specified, the Euclidean distance metric is used.

p_dist
[function, optional] Any function used to give weight to the distance between nodes when deciding whether or not they should be connected. `p_dist` was originally conceived as a probability density function giving the 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 `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 r^{-2} is used.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

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.

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], [2]

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)
```

5.11.3 navigable_small_world_graph

navigable_small_world_graph(*n*, *p=1*, *q=1*, *r=2*, *dim=2*, *seed=None*)

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 $i`thdirectededgefrom : math : `u$ 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.

p
[int] The diameter of short range connections. Each node is joined with every other node within this lattice distance.

q
[int] The number of long-range connections for each node.

r
[float] Exponent for decaying probability of connections. The probability of connecting to a node at lattice distance d is $1/d^r$.

dim
[int] Dimension of grid

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

References

[1]

5.11.4 random_geometric_graph

random_geometric_graph(*n*, *radius*, *dim=2*, *pos=None*, *p=2*, *seed=None*)

Returns a random geometric graph in the unit cube of dimensions `dim`.

The random geometric graph model places `n` nodes uniformly at random in the unit cube. Two nodes are joined by an edge if the distance between the nodes is at most `radius`.

Edges are determined using a KDTree when SciPy is available. This reduces the time complexity from $O(n^2)$ to $O(n)$.

Parameters

n
[int or iterable] Number of nodes or iterable of nodes

radius: float
Distance threshold value

dim
[int, optional] Dimension of graph

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 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 *Randomness*.

Returns

Graph
A 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.

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:

```
>>> import random
>>> n = 20
>>> pos = {i: (random.gauss(0, 2), random.gauss(0, 2)) for i in range(n)}
>>> G = nx.random_geometric_graph(n, 0.2, pos=pos)
```

References

[1]

Examples

Create a random geometric graph on twenty nodes where nodes are joined by an edge if their distance is at most 0.1:

```
>>> G = nx.random_geometric_graph(20, 0.1)
```

5.11.5 soft_random_geometric_graph

soft_random_geometric_graph(*n*, *radius*, *dim=2*, *pos=None*, *p=2*, *p_dist=None*, *seed=None*)

Returns a soft random geometric graph in the unit cube.

The soft random geometric graph [1] model places `n` nodes uniformly at random in the unit cube in dimension `dim`. Two nodes of distance, `dist`, computed by the `p`-Minkowski distance metric are joined by an edge with probability `p_dist` if the computed distance metric value of the nodes is at most `radius`, otherwise they are not joined.

Edges within `radius` of each other are determined using a KDTree when SciPy is available. This reduces the time complexity from $O(n^2)$ to $O(n)$.

Parameters

n
[int or iterable] Number of nodes or iterable of nodes

radius: float
Distance threshold value

dim
[int, optional] Dimension of graph

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 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 $\lambda = 1$.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

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.

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], [2]

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 $\lambda = 1$ if their Euclidean distance is at most 0.2.

5.11.6 thresholded_random_geometric_graph

thresholded_random_geometric_graph(*n*, *radius*, *theta*, *dim=2*, *pos=None*, *weight=None*, *p=2*, *seed=None*)

Returns a thresholded random geometric graph in the unit cube.

The thresholded random geometric graph [1] model places `n` nodes uniformly at random in the unit cube of dimensions `dim`. Each node `u` is assigned a weight w_u. Two nodes `u` and `v` are joined by an edge if they are within the maximum connection distance, `radius` computed by the `p`-Minkowski distance and the summation of weights w_u + w_v is greater than or equal to the threshold parameter `theta`.

Edges within `radius` of each other are determined using a KDTree when SciPy is available. This reduces the time complexity from $O(n^2)$ to $O(n)$.

Parameters

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 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 *Randomness*.

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.

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 $\lambda = 1$. 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]

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.

5.11.7 waxman_graph

waxman_graph(*n*, *beta=0.4*, *alpha=0.1*, *L=None*, *domain=(0, 0, 1, 1)*, *metric=None*, *seed=None*)

Returns a Waxman random graph.

The Waxman random graph model places `n` nodes uniformly at random in a rectangular domain. Each pair of nodes at distance `d` is joined by an edge with probability

$$p = \beta \exp(-d/\alpha L).$$

This function implements both Waxman models, using the `L` keyword argument.

- Waxman-1: if `L` is not specified, it is set to be the maximum distance between any pair of nodes.
- Waxman-2: if `L` is specified, the distance between a pair of nodes is chosen uniformly at random from the interval `[0, L]`.

Parameters

n
[int or iterable] Number of nodes or iterable of nodes

beta: float
Model parameter

alpha: float
Model parameter

L
[float, optional] Maximum distance between nodes. If not specified, the actual distance is calculated.

domain
[four-tuple of numbers, optional] Domain size, given as a tuple of the form `(x_min, y_min, x_max, y_max)`.

metric
[function] A metric on vectors of numbers (represented as lists or tuples). This must be a function that accepts two lists (or tuples) as input and yields a number as output. The function must also satisfy the four requirements of a metric. Specifically, if d is the function and x, y, and z are vectors in the graph, then d must satisfy

1. $d(x, y) \geq 0$,
2. $d(x, y) = 0$ if and only if $x = y$,
3. $d(x, y) = d(y, x)$,
4. $d(x, z) \leq d(x, y) + d(y, z)$.

If this argument is not specified, the Euclidean distance metric is used.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

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.

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]

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)
```

5.12 Line Graph

Functions for generating line graphs.

line_graph(G[, create_using])	Returns the line graph of the graph or digraph `G`.
inverse_line_graph(G)	Returns the inverse line graph of graph G.

5.12.1 line_graph

line_graph (*G*, *create_using=None*)

Returns the line graph of the graph or digraph `G`.

The line graph of a graph `G` has a node for each edge in `G` and an edge joining those nodes if the two edges in `G` share a common node. For directed graphs, nodes are adjacent exactly when the edges they represent form a directed path of length two.

The nodes of the line graph are 2-tuples of nodes in the original graph (or 3-tuples for multigraphs, with the key of the edge as the third element).

For information about self-loops and more discussion, see the **Notes** section below.

Parameters

G
[graph] A NetworkX Graph, DiGraph, MultiGraph, or MultiDigraph.

create_using
[NetworkX graph constructor, optional (default=nx.Graph)] Graph type to create. If graph instance, then cleared before populated.

Returns

L
[graph] The line graph of G.

Notes

Graph, node, and edge data are not propagated to the new graph. For undirected graphs, the nodes in G must be sortable, otherwise the constructed line graph may not be correct.

Self-loops in undirected graphs

For an undirected graph `G` without multiple edges, each edge can be written as a set `{u, v}`. Its line graph `L` has the edges of `G` as its nodes. If `x` and `y` are two nodes in `L`, then `{x, y}` is an edge in `L` if and only if the intersection of `x` and `y` is nonempty. Thus, the set of all edges is determined by the set of all pairwise intersections of edges in `G`.

Trivially, every edge in G would have a nonzero intersection with itself, and so every node in `L` should have a self-loop. This is not so interesting, and the original context of line graphs was with simple graphs, which had no self-loops or multiple edges. The line graph was also meant to be a simple graph and thus, self-loops in `L` are not part of the standard definition of a line graph. In a pairwise intersection matrix, this is analogous to excluding the diagonal entries from the line graph definition.

Self-loops and multiple edges in `G` add nodes to `L` in a natural way, and do not require any fundamental changes to the definition. It might be argued that the self-loops we excluded before should now be included. However, the self-loops are still "trivial" in some sense and thus, are usually excluded.

Self-loops in directed graphs

For a directed graph `G` without multiple edges, each edge can be written as a tuple `(u, v)`. Its line graph `L` has the edges of `G` as its nodes. If `x` and `y` are two nodes in `L`, then `(x, y)` is an edge in `L` if and only if the tail of `x` matches the head of `y`, for example, if `x = (a, b)` and `y = (b, c)` for some vertices `a`, `b`, and `c` in `G`.

Due to the directed nature of the edges, it is no longer the case that every edge in `G` should have a self-loop in `L`. Now, the only time self-loops arise is if a node in `G` itself has a self-loop. So such self-loops are no longer "trivial" but instead, represent essential features of the topology of `G`. For this reason, the historical development of line digraphs is such that self-loops are included. When the graph `G` has multiple edges, once again only superficial changes are required to the definition.

References

- Harary, Frank, and Norman, Robert Z., "Some properties of line digraphs", Rend. Circ. Mat. Palermo, II. Ser. 9 (1960), 161–168.
- Hemminger, R. L.; Beineke, L. W. (1978), "Line graphs and line digraphs", in Beineke, L. W.; Wilson, R. J., Selected Topics in Graph Theory, Academic Press Inc., pp. 271–305.

Examples

```
>>> G = nx.star_graph(3)
>>> L = nx.line_graph(G)
>>> print(sorted(map(sorted, L.edges())))  # makes a 3-clique, K3
[[(0, 1), (0, 2)], [(0, 1), (0, 3)], [(0, 2), (0, 3)]]
```

Edge attributes from `G` are not copied over as node attributes in `L`, but attributes can be copied manually:

```
>>> G = nx.path_graph(4)
>>> G.add_edges_from((u, v, {"tot": u+v}) for u, v in G.edges)
>>> G.edges(data=True)
EdgeDataView([(0, 1, {'tot': 1}), (1, 2, {'tot': 3}), (2, 3, {'tot': 5})])
>>> H = nx.line_graph(G)
```

(continues on next page)

(continued from previous page)

```
>>> H.add_nodes_from((node, G.edges[node]) for node in H)
>>> H.nodes(data=True)
NodeDataView({(0, 1): {'tot': 1}, (2, 3): {'tot': 5}, (1, 2): {'tot': 3}})
```

5.12.2 inverse_line_graph

inverse_line_graph(*G*)

Returns the inverse line graph of graph G.

If H is a graph, and G is the line graph of H, such that G = L(H). Then H is the inverse line graph of G.

Not all graphs are line graphs and these do not have an inverse line graph. In these cases this function raises a NetworkXError.

Parameters

G
[graph] A NetworkX Graph

Returns

H
[graph] The inverse line graph of G.

Raises

NetworkXNotImplemented
If G is directed or a multigraph

NetworkXError
If G is not a line graph

Notes

This is an implementation of the Roussopoulos algorithm.

If G consists of multiple components, then the algorithm doesn't work. You should invert every component seperately:

```
>>> K5 = nx.complete_graph(5)
>>> P4 = nx.Graph([("a", "b"), ("b", "c"), ("c", "d")])
>>> G = nx.union(K5, P4)
>>> root_graphs = []
>>> for comp in nx.connected_components(G):
...     root_graphs.append(nx.inverse_line_graph(G.subgraph(comp)))
>>> len(root_graphs)
2
```

References

- Roussopolous, N, "A max {m, n} algorithm for determining the graph H from its line graph G", Information Processing Letters 2, (1973), 108–112.

5.13 Ego Graph

Ego graph.

ego_graph(G, n[, radius, center, ...])	Returns induced subgraph of neighbors centered at node n within a given radius.

5.13.1 ego_graph

ego_graph(*G*, *n*, *radius=1*, *center=True*, *undirected=False*, *distance=None*)

Returns induced subgraph of neighbors centered at node n within a given radius.

Parameters

G
[graph] A NetworkX Graph or DiGraph

n
[node] A single node

radius
[number, optional] Include all neighbors of distance<=radius from n.

center
[bool, optional] If False, do not include center node in graph

undirected
[bool, optional] If True use both in- and out-neighbors of directed graphs.

distance
[key, optional] Use specified edge data key as distance. For example, setting distance='weight' will use the edge weight to measure the distance from the node n.

Notes

For directed graphs D this produces the "out" neighborhood or successors. If you want the neighborhood of predecessors first reverse the graph with D.reverse(). If you want both directions use the keyword argument undirected=True.

Node, edge, and graph attributes are copied to the returned subgraph.

5.14 Stochastic

Functions for generating stochastic graphs from a given weighted directed graph.

stochastic_graph(G[, copy, weight])	Returns a right-stochastic representation of directed graph G.

5.14.1 stochastic_graph

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 *DiGraph* or *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.

5.15 AS graph

Generates graphs resembling the Internet Autonomous System network

random_internet_as_graph(n[, seed])	Generates a random undirected graph resembling the Internet AS network

5.15.1 random_internet_as_graph

random_internet_as_graph(*n*, *seed=None*)

Generates a random undirected graph resembling the Internet AS network

Parameters

n: integer in [1000, 10000]
Number of graph nodes

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

G: Networkx Graph object
A randomly generated undirected graph

Notes

This algorithm returns an undirected graph resembling the Internet Autonomous System (AS) network, it uses the approach by Elmokashfi et al. [1] and it grants the properties described in the related paper [1].

Each node models an autonomous system, with an attribute 'type' specifying its kind; tier-1 (T), mid-level (M), customer (C) or content-provider (CP). Each edge models an ADV communication link (hence, bidirectional) with attributes:

- type: transit|peer, the kind of commercial agreement between nodes;
- customer: <node id>, the identifier of the node acting as customer ('none' if type is peer).

References

[1]

5.16 Intersection

Generators for random intersection graphs.

uniform_random_intersection_graph(n, m, p[, ...])	Returns a uniform random intersection graph.
k_random_intersection_graph(n, m, k[, seed])	Returns a intersection graph with randomly chosen attribute sets for each node that are of equal size (k).
general_random_intersection_graph(n, m, p[, ...])	Returns a random intersection graph with independent probabilities for connections between node and attribute sets.

5.16.1 uniform_random_intersection_graph

uniform_random_intersection_graph(*n*, *m*, *p*, *seed=None*)

Returns a uniform random intersection graph.

Parameters

n
[int] The number of nodes in the first bipartite set (nodes)

m
[int] The number of nodes in the second bipartite set (attributes)

p
[float] Probability of connecting nodes between bipartite sets

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

See also:

`gnp_random_graph`

References

[1], [2]

5.16.2 k_random_intersection_graph

k_random_intersection_graph(*n*, *m*, *k*, *seed=None*)

Returns a intersection graph with randomly chosen attribute sets for each node that are of equal size (k).

Parameters

n
[int] The number of nodes in the first bipartite set (nodes)

m
[int] The number of nodes in the second bipartite set (attributes)

k
[float] Size of attribute set to assign to each node.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

See also:

`gnp_random_graph`, *`uniform_random_intersection_graph`*

References

[1]

5.16.3 general_random_intersection_graph

general_random_intersection_graph(*n*, *m*, *p*, *seed=None*)

Returns a random intersection graph with independent probabilities for connections between node and attribute sets.

Parameters

n
[int] The number of nodes in the first bipartite set (nodes)

m
[int] The number of nodes in the second bipartite set (attributes)

p
[list of floats of length m] Probabilities for connecting nodes to each attribute

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

See also:

gnp_random_graph, *uniform_random_intersection_graph*

References

[1]

5.17 Social Networks

Famous social networks.

karate_club_graph()	Returns Zachary's Karate Club graph.
davis_southern_women_graph()	Returns Davis Southern women social network.
florentine_families_graph()	Returns Florentine families graph.
les_miserables_graph()	Returns coappearance network of characters in the novel Les Miserables.

5.17.1 karate_club_graph

karate_club_graph()

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.

References

[1]

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'
```

5.17.2 davis_southern_women_graph

davis_southern_women_graph()

Returns Davis Southern women social network.

This is a bipartite graph.

References

[1]

5.17.3 florentine_families_graph

florentine_families_graph()

Returns Florentine families graph.

References

[1]

5.17.4 les_miserables_graph

les_miserables_graph()

Returns coappearance network of characters in the novel Les Miserables.

References

[1]

5.18 Community

Generators for classes of graphs used in studying social networks.

caveman_graph(l, k)	Returns a caveman graph of `l` cliques of size `k`.
connected_caveman_graph(l, k)	Returns a connected caveman graph of `l` cliques of size `k`.
gaussian_random_partition_graph(n, s, v, ...)	Generate a Gaussian random partition graph.
LFR_benchmark_graph(n, tau1, tau2, mu[, ...])	Returns the LFR benchmark graph.
planted_partition_graph(l, k, p_in, p_out[, ...])	Returns the planted l-partition graph.
random_partition_graph(sizes, p_in, p_out[, ...])	Returns the random partition graph with a partition of sizes.
relaxed_caveman_graph(l, k, p[, seed])	Returns a relaxed caveman graph.
ring_of_cliques(num_cliques, clique_size)	Defines a "ring of cliques" graph.
stochastic_block_model(sizes, p[, nodelist, ...])	Returns a stochastic block model graph.
windmill_graph(n, k)	Generate a windmill graph.

5.18.1 caveman_graph

caveman_graph(*l*, *k*)

Returns a caveman graph of `l` cliques of size `k`.

Parameters

l
[int] Number of cliques

k
[int] Size of cliques

Returns

G
[NetworkX Graph] caveman graph

See also:

connected_caveman_graph

Notes

This returns an undirected graph, it can be converted to a directed graph using `nx.to_directed()`, or a multigraph using `nx.MultiGraph(nx.caveman_graph(l, k))`. Only the undirected version is described in [1] and it is unclear which of the directed generalizations is most useful.

References

[1]

Examples

```
>>> G = nx.caveman_graph(3, 3)
```

5.18.2 connected_caveman_graph

connected_caveman_graph(*l*, *k*)

Returns a connected caveman graph of `l` cliques of size `k`.

The connected caveman graph is formed by creating `n` cliques of size `k`, then a single edge in each clique is rewired to a node in an adjacent clique.

Parameters

l
[int] number of cliques

k
[int] size of cliques (k at least 2 or NetworkXError is raised)

Returns

G
[NetworkX Graph] connected caveman graph

Raises

NetworkXError
If the size of cliques `k` is smaller than 2.

Notes

This returns an undirected graph, it can be converted to a directed graph using `nx.to_directed()`, or a multigraph using `nx.MultiGraph(nx.caveman_graph(l, k))`. Only the undirected version is described in [1] and it is unclear which of the directed generalizations is most useful.

References

[1]

Examples

```
>>> G = nx.connected_caveman_graph(3, 3)
```

5.18.3 gaussian_random_partition_graph

gaussian_random_partition_graph(*n*, *s*, *v*, *p_in*, *p_out*, *directed=False*, *seed=None*)

Generate a Gaussian random partition graph.

A Gaussian random partition graph is created by creating k partitions each with a size drawn from a normal distribution with mean s and variance s/v. Nodes are connected within clusters with probability p_in and between clusters with probability p_out[1]

Parameters

n
[int] Number of nodes in the graph

s
[float] Mean cluster size

v
[float] Shape parameter. The variance of cluster size distribution is s/v.

p_in
[float] Probabilty of intra cluster connection.

p_out
[float] Probability of inter cluster connection.

directed
[boolean, optional default=False] Whether to create a directed graph or not

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

G
[NetworkX Graph or DiGraph] gaussian random partition graph

Raises

NetworkXError
If s is > n If p_in or p_out is not in [0,1]

See also:

random_partition_graph

Notes

Note the number of partitions is dependent on s,v and n, and that the last partition may be considerably smaller, as it is sized to simply fill out the nodes [1]

References

[1]

Examples

```
>>> G = nx.gaussian_random_partition_graph(100, 10, 10, 0.25, 0.1)
>>> len(G)
100
```

5.18.4 LFR_benchmark_graph

LFR_benchmark_graph(*n*, *tau1*, *tau2*, *mu*, *average_degree=None*, *min_degree=None*, *max_degree=None*, *min_community=None*, *max_community=None*, *tol=1e-07*, *max_iters=500*, *seed=None*)

Returns the LFR benchmark graph.

This algorithm proceeds as follows:

1) Find a degree sequence with a power law distribution, and minimum value `min_degree`, which has approximate average degree `average_degree`. This is accomplished by either

   a) specifying `min_degree` and not `average_degree`,

   b) specifying `average_degree` and not `min_degree`, in which case a suitable minimum degree will be found.

   `max_degree` can also be specified, otherwise it will be set to `n`. Each node u will have $\mu \mathrm{deg}(u)$ edges joining it to nodes in communities other than its own and $(1 - \mu)\mathrm{deg}(u)$ edges joining it to nodes in its own community.

2) Generate community sizes according to a power law distribution with exponent `tau2`. If `min_community` and `max_community` are not specified they will be selected to be `min_degree` and `max_degree`, respectively. Community sizes are generated until the sum of their sizes equals `n`.

3) Each node will be randomly assigned a community with the condition that the community is large enough for the node's intra-community degree, $(1 - \mu)\mathrm{deg}(u)$ as described in step 2. If a community grows too large, a random node will be selected for reassignment to a new community, until all nodes have been assigned a community.

4) Each node u then adds $(1 - \mu)\deg(u)$ intra-community edges and $\mu\deg(u)$ inter-community edges.

Parameters

n
[int] Number of nodes in the created graph.

tau1
[float] Power law exponent for the degree distribution of the created graph. This value must be strictly greater than one.

tau2
[float] Power law exponent for the community size distribution in the created graph. This value must be strictly greater than one.

mu
[float] Fraction of inter-community edges incident to each node. This value must be in the interval [0, 1].

average_degree
[float] Desired average degree of nodes in the created graph. This value must be in the interval [0, *n*]. Exactly one of this and `min_degree` must be specified, otherwise a `NetworkXError` is raised.

min_degree
[int] Minimum degree of nodes in the created graph. This value must be in the interval [0, *n*]. Exactly one of this and `average_degree` must be specified, otherwise a `NetworkXError` is raised.

max_degree
[int] Maximum degree of nodes in the created graph. If not specified, this is set to `n`, the total number of nodes in the graph.

min_community
[int] Minimum size of communities in the graph. If not specified, this is set to `min_degree`.

max_community
[int] Maximum size of communities in the graph. If not specified, this is set to `n`, the total number of nodes in the graph.

tol
[float] Tolerance when comparing floats, specifically when comparing average degree values.

max_iters
[int] 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 *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 $\{1, \ldots, n\}$.
- `min_community` and `max_community` must be in $\{0, \ldots, n\}$.

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.

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], [2]

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}
```

5.18.5 planted_partition_graph

planted_partition_graph(*l*, *k*, *p_in*, *p_out*, *seed=None*, *directed=False*)

Returns the planted l-partition graph.

This model partitions a graph with n=l*k vertices in l groups with k vertices each. Vertices of the same group are linked with a probability p_in, and vertices of different groups are linked with probability p_out.

Parameters

l
[int] Number of groups

k
[int] Number of vertices in each group

p_in
[float] probability of connecting vertices within a group

p_out
[float] probability of connected vertices between groups

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

directed
[bool,optional (default=False)] If True return a directed graph

Returns

G
[NetworkX Graph or DiGraph] planted l-partition graph

Raises

NetworkXError
If p_in,p_out are not in [0,1] or

See also:

random_partition_model

References

[1], [2]

Examples

```
>>> G = nx.planted_partition_graph(4, 3, 0.5, 0.1, seed=42)
```

5.18.6 random_partition_graph

random_partition_graph(*sizes*, *p_in*, *p_out*, *seed=None*, *directed=False*)

Returns the random partition graph with a partition of sizes.

A partition graph is a graph of communities with sizes defined by s in sizes. Nodes in the same group are connected with probability p_in and nodes of different groups are connected with probability p_out.

Parameters

sizes
[list of ints] Sizes of groups

p_in
[float] probability of edges with in groups

p_out
[float] probability of edges between groups

directed
[boolean optional, default=False] Whether to create a directed graph

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

G
[NetworkX Graph or DiGraph] random partition graph of size sum(gs)

Raises

NetworkXError
If p_in or p_out is not in [0,1]

Notes

This is a generalization of the planted-l-partition described in [1]. It allows for the creation of groups of any size.

The partition is store as a graph attribute 'partition'.

References

[1]

Examples

```
>>> G = nx.random_partition_graph([10, 10, 10], 0.25, 0.01)
>>> len(G)
30
>>> partition = G.graph["partition"]
>>> len(partition)
3
```

5.18.7 relaxed_caveman_graph

relaxed_caveman_graph(*l*, *k*, *p*, *seed=None*)

Returns a relaxed caveman graph.

A relaxed caveman graph starts with `l` cliques of size `k`. Edges are then randomly rewired with probability `p` to link different cliques.

Parameters

l
[int] Number of groups

k
[int] Size of cliques

p
[float] Probabilty of rewiring each edge.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

G
[NetworkX Graph] Relaxed Caveman Graph

Raises

NetworkXError
If p is not in [0,1]

References

[1]

Examples

```
>>> G = nx.relaxed_caveman_graph(2, 3, 0.1, seed=42)
```

5.18.8 ring_of_cliques

ring_of_cliques(*num_cliques*, *clique_size*)

Defines a "ring of cliques" graph.

A ring of cliques graph is consisting of cliques, connected through single links. Each clique is a complete graph.

Parameters

num_cliques
[int] Number of cliques

clique_size
[int] Size of cliques

Returns

G
[NetworkX Graph] ring of cliques graph

Raises

NetworkXError
If the number of cliques is lower than 2 or if the size of cliques is smaller than 2.

See also:

connected_caveman_graph

Notes

The *connected_caveman_graph* graph removes a link from each clique to connect it with the next clique. Instead, the *ring_of_cliques* graph simply adds the link without removing any link from the cliques.

Examples

```
>>> G = nx.ring_of_cliques(8, 4)
```

5.18.9 stochastic_block_model

stochastic_block_model(*sizes*, *p*, *nodelist=None*, *seed=None*, *directed=False*, *selfloops=False*, *sparse=True*)

Returns a stochastic block model graph.

This model partitions the nodes in blocks of arbitrary sizes, and places edges between pairs of nodes independently, with a probability that depends on the blocks.

Parameters

sizes
[list of ints] Sizes of blocks

p
[list of list of floats] Element (r,s) gives the density of edges going from the nodes of group r to nodes of group s. p must match the number of groups (len(sizes) == len(p)), and it must be symmetric if the graph is undirected.

nodelist
[list, optional] The block tags are assigned according to the node identifiers in nodelist. If nodelist is None, then the ordering is the range [0,sum(sizes)-1].

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

directed
[boolean optional, default=False] Whether to create a directed graph or not.

selfloops
[boolean optional, default=False] Whether to include self-loops or not.

sparse: boolean optional, default=True
Use the sparse heuristic to speed up the generator.

Returns

g
[NetworkX Graph or DiGraph] Stochastic block model graph of size sum(sizes)

Raises

NetworkXError
If probabilities are not in [0,1]. If the probability matrix is not square (directed case). If the probability matrix is not symmetric (undirected case). If the sizes list does not match nodelist or the probability matrix. If nodelist contains duplicate.

See also:

random_partition_graph
planted_partition_graph
gaussian_random_partition_graph
gnp_random_graph

References

[1]

Examples

```
>>> sizes = [75, 75, 300]
>>> probs = [[0.25, 0.05, 0.02], [0.05, 0.35, 0.07], [0.02, 0.07, 0.40]]
>>> g = nx.stochastic_block_model(sizes, probs, seed=0)
>>> len(g)
450
>>> H = nx.quotient_graph(g, g.graph["partition"], relabel=True)
>>> for v in H.nodes(data=True):
...     print(round(v[1]["density"], 3))
...
0.245
0.348
0.405
>>> for v in H.edges(data=True):
...     print(round(1.0 * v[2]["weight"] / (sizes[v[0]] * sizes[v[1]]), 3))
...
0.051
0.022
0.07
```

5.18.10 windmill_graph

windmill_graph(*n*, *k*)

Generate a windmill graph. A windmill graph is a graph of `n` cliques each of size `k` that are all joined at one node. It can be thought of as taking a disjoint union of `n` cliques of size `k`, selecting one point from each, and contracting all of the selected points. Alternatively, one could generate `n` cliques of size `k-1` and one node that is connected to all other nodes in the graph.

Parameters

n
[int] Number of cliques

k
[int] Size of cliques

Returns

G
[NetworkX Graph] windmill graph with n cliques of size k

Raises

NetworkXError
If the number of cliques is less than two If the size of the cliques are less than two

Notes

The node labeled `0` will be the node connected to all other nodes. Note that windmill graphs are usually denoted `Wd(k,n)`, so the parameters are in the opposite order as the parameters of this method.

Examples

```
>>> G = nx.windmill_graph(4, 5)
```

5.19 Spectral

Generates graphs with a given eigenvector structure

spectral_graph_forge(G, alpha[, ...])	Returns a random simple graph with spectrum resembling that of `G`

5.19.1 spectral_graph_forge

spectral_graph_forge(*G*, *alpha*, *transformation='identity'*, *seed=None*)

Returns a random simple graph with spectrum resembling that of `G`

This algorithm, called Spectral Graph Forge (SGF), computes the eigenvectors of a given graph adjacency matrix, filters them and builds a random graph with a similar eigenstructure. SGF has been proved to be particularly useful for synthesizing realistic social networks and it can also be used to anonymize graph sensitive data.

Parameters

G
[Graph]

alpha
[float] Ratio representing the percentage of eigenvectors of G to consider, values in [0,1].

transformation
[string, optional] Represents the intended matrix linear transformation, possible values are 'identity' and 'modularity'

seed
[integer, random_state, or None (default)] Indicator of numpy random number generation state. See *Randomness*.

Returns

H
[Graph] A graph with a similar eigenvector structure of the input one.

Raises

NetworkXError
If transformation has a value different from 'identity' or 'modularity'

Notes

Spectral Graph Forge (SGF) generates a random simple graph resembling the global properties of the given one. It leverages the low-rank approximation of the associated adjacency matrix driven by the *alpha* precision parameter. SGF preserves the number of nodes of the input graph and their ordering. This way, nodes of output graphs resemble the properties of the input one and attributes can be directly mapped.

It considers the graph adjacency matrices which can optionally be transformed to other symmetric real matrices (currently transformation options include *identity* and *modularity*). The *modularity* transformation, in the sense of Newman's modularity matrix allows the focusing on community structure related properties of the graph.

SGF applies a low-rank approximation whose fixed rank is computed from the ratio *alpha* of the input graph adjacency matrix dimension. This step performs a filtering on the input eigenvectors similar to the low pass filtering common in telecommunications.

The filtered values (after truncation) are used as input to a Bernoulli sampling for constructing a random adjacency matrix.

References

Examples

```
>>> G = nx.karate_club_graph()
>>> H = nx.spectral_graph_forge(G, 0.3)
>>>
```

5.20 Trees

Functions for generating trees.

random_tree(n[, seed, create_using])	Returns a uniformly random tree on `n` nodes.
prefix_tree(paths)	Creates a directed prefix tree from a list of paths.

5.20.1 random_tree

random_tree(*n*, *seed=None*, *create_using=None*)

Returns a uniformly random tree on `n` nodes.

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 *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 `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)
>>> print(nx.forest_str(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)
>>> print(nx.forest_str(tree))
╙── 0
    ├── 3
    └── 4
        ├── 6
        │   ├── 1
```

(continues on next page)

(continued from previous page)

```
    ├─ 2
    └─ 7
        └─ 8
            └─ 5
└─ 9
```

5.20.2 prefix_tree

prefix_tree(*paths*)

Creates a directed prefix tree from a list of paths.

Usually the paths are described as strings or lists of integers.

A "prefix tree" represents the prefix structure of the strings. Each node represents a prefix of some string. The root represents the empty prefix with children for the single letter prefixes which in turn have children for each double letter prefix starting with the single letter corresponding to the parent node, and so on.

More generally the prefixes do not need to be strings. A prefix refers to the start of a sequence. The root has children for each one element prefix and they have children for each two element prefix that starts 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']
```

5.21 Non Isomorphic Trees

Implementation of the Wright, Richmond, Odlyzko and McKay (WROM) algorithm for the enumeration of all non-isomorphic free trees of a given order. Rooted trees are represented by level sequences, i.e., lists in which the i-th element specifies the distance of vertex i to the root.

nonisomorphic_trees(order[, create])	Returns a list of nonisomporphic trees
number_of_nonisomorphic_trees(order)	Returns the number of nonisomorphic trees

5.21.1 nonisomorphic_trees

nonisomorphic_trees(*order*, *create='graph'*)

Returns a list of nonisomporphic trees

Parameters

order
[int] order of the desired tree(s)

create
[graph or matrix (default="Graph)] If graph is selected a list of trees will be returned, if matrix is selected a list of adjancency matrix will be returned

Returns

G
[List of NetworkX Graphs]

M
[List of Adjacency matrices]

5.21.2 number_of_nonisomorphic_trees

number_of_nonisomorphic_trees(*order*)

Returns the number of nonisomorphic trees

Parameters

order
[int] order of the desired tree(s)

Returns

length
[Number of nonisomorphic graphs for the given order]

5.22 Triads

Functions that generate the triad graphs, that is, the possible digraphs on three nodes.

triad_graph(triad_name)	Returns the triad graph with the given name.

5.22.1 triad_graph

triad_graph(*triad_name*)

Returns the triad graph with the given name.

Each string in the following tuple is a valid triad name:

```
('003', '012', '102', '021D', '021U', '021C', '111D', '111U',
 '030T', '030C', '201', '120D', '120U', '120C', '210', '300')
```

Each triad name corresponds to one of the possible valid digraph on three nodes.

Parameters

triad_name
[string] The name of a triad, as described above.

Returns

DiGraph
The digraph on three nodes with the given name. The nodes of the graph are the single-character strings 'a', 'b', and 'c'.

Raises

ValueError
If `triad_name` is not the name of a triad.

See also:

`triadic_census`

5.23 Joint Degree Sequence

Generate graphs with a given joint degree and directed joint degree

is_valid_joint_degree(joint_degrees)	Checks whether the given joint degree dictionary is realizable.
joint_degree_graph(joint_degrees[, seed])	Generates a random simple graph with the given joint degree dictionary.
is_valid_directed_joint_degree(in_degrees, ...)	Checks whether the given directed joint degree input is realizable
directed_joint_degree_graph(in_degrees, ...)	Generates a random simple directed graph with the joint degree.

5.23.1 is_valid_joint_degree

is_valid_joint_degree(*joint_degrees*)

Checks whether the given joint degree dictionary is realizable.

A *joint degree dictionary* is a dictionary of dictionaries, in which entry `joint_degrees[k][l]` is an integer representing the number of edges joining nodes of degree *k* with nodes of degree *l*. Such a dictionary is realizable as a simple graph if and only if the following conditions are satisfied.

- each entry must be an integer,
- the total number of nodes of degree *k*, computed by `sum(joint_degrees[k].values()) / k`, must be an integer,
- the total number of edges joining nodes of degree *k* with nodes of degree *l* cannot exceed the total number of possible edges,
- each diagonal entry `joint_degrees[k][k]` must be even (this is a convention assumed by the *joint_degree_graph()* function).

Parameters

joint_degrees
[dictionary of dictionary of integers] A joint degree dictionary in which entry `joint_degrees[k][l]` is the number of edges joining nodes of degree *k* with nodes of degree *l*.

Returns

bool
Whether the given joint degree dictionary is realizable as a simple graph.

References

[1], [2]

5.23.2 joint_degree_graph

joint_degree_graph(*joint_degrees*, *seed=None*)

Generates a random simple graph with the given joint degree dictionary.

Parameters

joint_degrees
: [dictionary of dictionary of integers] A joint degree dictionary in which entry `joint_degrees[k][l]` is the number of edges joining nodes of degree *k* with nodes of degree *l*.

seed
: [integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

G
: [Graph] A graph with the specified joint degree dictionary.

Raises

NetworkXError
: If *joint_degrees* dictionary is not realizable.

Notes

In each iteration of the "while loop" the algorithm picks two disconnected nodes *v* and *w*, of degree *k* and *l* correspondingly, for which `joint_degrees[k][l]` has not reached its target yet. It then adds edge (*v*, *w*) and increases the number of edges in graph G by one.

The intelligence of the algorithm lies in the fact that it is always possible to add an edge between such disconnected nodes *v* and *w*, even if one or both nodes do not have free stubs. That is made possible by executing a "neighbor switch", an edge rewiring move that releases a free stub while keeping the joint degree of G the same.

The algorithm continues for E (number of edges) iterations of the "while loop", at the which point all entries of the given `joint_degrees[k][l]` have reached their target values and the construction is complete.

References

Examples

```
>>> joint_degrees = {
...     1: {4: 1},
...     2: {2: 2, 3: 2, 4: 2},
...     3: {2: 2, 4: 1},
...     4: {1: 1, 2: 2, 3: 1},
... }
>>> G = nx.joint_degree_graph(joint_degrees)
>>>
```

5.23.3 is_valid_directed_joint_degree

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 (seconnd 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.

5.23.4 directed_joint_degree_graph

directed_joint_degree_graph(*in_degrees*, *out_degrees*, *nkk*, *seed=None*)

Generates a random simple directed graph with the joint degree.

Parameters

degree_seq
[list of tuples (of size 3)] degree sequence contains tuples of nodes with node id, in degree and out degree.

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.

seed
[hashable object, optional] Seed for random number generator.

Returns

G
[Graph] A directed graph with the specified inputs.

Raises

NetworkXError
If degree_seq and nkk are not realizable as a simple directed graph.

Notes

Similarly to the undirected version: In each iteration of the "while loop" the algorithm picks two disconnected nodes v and w, of degree k and l correspondingly, for which nkk[k][l] has not reached its target yet i.e. (for given k,l): n_edges_add < nkk[k][l]. It then adds edge (v,w) and always increases the number of edges in graph G by one.

The intelligence of the algorithm lies in the fact that it is always possible to add an edge between disconnected nodes v and w, for which nkk[degree(v)][degree(w)] has not reached its target, even if one or both nodes do not have free stubs. If either node v or w does not have a free stub, we perform a "neighbor switch", an edge rewiring move that releases a free stub while keeping nkk the same.

The difference for the directed version lies in the fact that neighbor switches might not be able to rewire, but in these cases unsaturated nodes can be reassigned to use instead, see [1] for detailed description and proofs.

The algorithm continues for E (number of edges in the graph) iterations of the "while loop", at which point all entries of the given nkk[k][l] have reached their target values and the construction is complete.

References

[1] B. Tillman, A. Markopoulou, C. T. Butts & M. Gjoka,
"Construction of Directed 2K Graphs". In Proc. of KDD 2017.

Examples

```
>>> in_degrees = [0, 1, 1, 2]
>>> out_degrees = [1, 1, 1, 1]
>>> nkk = {1: {1: 2, 2: 2}}
>>> G = nx.directed_joint_degree_graph(in_degrees, out_degrees, nkk)
>>>
```

5.24 Mycielski

Functions related to the Mycielski Operation and the Mycielskian family of graphs.

mycielskian(G[, iterations])	Returns the Mycielskian of a simple, undirected graph G
mycielski_graph(n)	Generator for the n_th Mycielski Graph.

5.24.1 mycielskian

mycielskian(*G*, *iterations=1*)

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, $G = (V, E)$, constructs a new graph with $2|V| + 1$ nodes and $3|E| + |V|$ edges.

The construction is as follows:

Let $V = 0, ..., n - 1$. Construct another vertex set $U = n, ..., 2n$ and a vertex, `w`. Construct a new graph, `M`, with vertices $U \bigcup V \bigcup w$. For edges, $(u, v) \in E$ add edges $(u, v), (u, v + n)$, and $(u + n, v)$ to M. Finally, for all vertices $u \in U$, add edge (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.

5.24.2 mycielski_graph

mycielski_graph(*n*)

Generator for the n_th Mycielski Graph.

The Mycielski family of graphs is an infinite set of graphs. M_1 is the singleton graph, M_2 is two vertices with an edge, and, for $i > 2$, M_i is the Mycielskian of 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 P_2 graph, but rather the P_2 graph with an extra, isolated vertex. The second Mycielski graph is the P_2 graph, so the first two are hard coded. The remaining graphs are generated using the Mycielski operation.

5.25 Harary Graph

Generators for Harary graphs

This module gives two generators for the Harary graph, which was introduced by the famous mathematician Frank Harary in his 1962 work [H]. The first generator gives the Harary graph that maximizes the node connectivity with given number of nodes and given number of edges. The second generator gives the Harary graph that minimizes the number of edges in the graph with given node connectivity and number of nodes.

5.25.1 References

hnm_harary_graph(n, m[, create_using])	Returns the Harary graph with given numbers of nodes and edges.
hkn_harary_graph(k, n[, create_using])	Returns the Harary graph with given node connectivity and node number.

5.25.2 hnm_harary_graph

hnm_harary_graph(*n*, *m*, *create_using=None*)

Returns the Harary graph with given numbers of nodes and edges.

The Harary graph $H_{n,m}$ is the graph that maximizes node connectivity with n nodes and m edges.

This maximum node connectivity is known to be floor($2m/n$). [1]

Parameters

n: integer
The number of nodes the generated graph is to contain

m: integer
The number of edges the generated graph is to contain

create_using
[NetworkX graph constructor, optional Graph type] to create (default=nx.Graph). If graph instance, then cleared before populated.

Returns

NetworkX graph
The Harary graph $H_{n,m}$.

See also:

hkn_harary_graph

Notes

This algorithm runs in $O(m)$ time. It is implemented by following the Reference [2].

References

[1], [2]

5.25.3 hkn_harary_graph

hkn_harary_graph(*k*, *n*, *create_using=None*)

Returns the Harary graph with given node connectivity and node number.

The Harary graph $H_{k,n}$ is the graph that minimizes the number of edges needed with given node connectivity k and node number n.

This smallest number of edges is known to be ceil($kn/2$) [1].

Parameters

k: integer
The node connectivity of the generated graph

n: integer
The number of nodes the generated graph is to contain

create_using
[NetworkX graph constructor, optional Graph type] to create (default=nx.Graph). If graph instance, then cleared before populated.

Returns

NetworkX graph
The Harary graph $H_{k,n}$.

See also:

hnm_harary_graph

Notes

This algorithm runs in $O(kn)$ time. It is implemented by following the Reference [2].

References

[1], [2]

5.26 Cographs

Generators for cographs

A cograph is a graph containing no path on four vertices. Cographs or P_4-free graphs can be obtained from a single vertex by disjoint union and complementation operations.

5.26.1 References

random_cograph(n[, seed])	Returns a random cograph with 2^n nodes.

5.26.2 random_cograph

random_cograph(*n*, *seed=None*)

Returns a random cograph with 2^n nodes.

A cograph is a graph containing no path on four vertices. Cographs or P_4-free graphs can be obtained from a single vertex by disjoint union and complementation operations.

This generator starts off from a single vertex and performes disjoint union and full join operations on itself. The decision on which operation will take place is random.

Parameters

n
[int] The order of the cograph.

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

G
[A random graph containing no path on four vertices.]

See also:

full_join
union

References

[1]

5.27 Interval Graph

Generators for interval graph.

interval_graph(intervals)	Generates an interval graph for a list of intervals given.

5.27.1 interval_graph

interval_graph (*intervals*)

Generates an interval graph for a list of intervals given.

In graph theory, an interval graph is an undirected graph formed from a set of closed intervals on the real line, with a vertex for each interval and an edge between vertices whose intervals intersect. It is the intersection graph of the intervals.

More information can be found at: https://en.wikipedia.org/wiki/Interval_graph

Parameters

intervals
[a sequence of intervals, say (l, r) where l is the left end,]

and r is the right end of the closed interval.

Returns

G
[networkx graph]

Raises

TypeError
if `intervals` contains None or an element which is not collections.abc.Sequence or not a length of 2.

ValueError
if `intervals` contains an interval such that min1 > max1 where min1,max1 = interval

Examples

```
>>> intervals = [(-2, 3), [1, 4], (2, 3), (4, 6)]
>>> G = nx.interval_graph(intervals)
>>> sorted(G.edges)
[((-2, 3), (1, 4)), ((-2, 3), (2, 3)), ((1, 4), (2, 3)), ((1, 4), (4, 6))]
```

5.28 Sudoku

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 2 5 9 7 1	3 7 1 8 4 9 2 6 5	2 5 9 7 6 1 8 4 3
4 3 6 1 9 8 2 5 7	1 9 2 6 5 7 4 8 3	5 8 7 4 3 2 9 1 6
6 8 9 7 1 3 5 4 2	7 3 4 5 2 8 9 1 6	1 2 5 6 9 4 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.

5.28.1 References

sudoku_graph([n])	Returns the n-Sudoku graph.

5.28.2 sudoku_graph

sudoku_graph(*n=3*)

Returns the n-Sudoku graph. The default value of n is 3.

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.

Parameters

n: integer
The order of the Sudoku graph, equal to the square root of the number of rows. The default is 3.

Returns

NetworkX graph
The n-Sudoku graph Sud(n).

References

[1], [2], [3]

Examples

```
>>> G = nx.sudoku_graph()
>>> G.number_of_nodes()
81
>>> G.number_of_edges()
810
>>> sorted(G.neighbors(42))
[6, 15, 24, 33, 34, 35, 36, 37, 38, 39, 40, 41, 43, 44, 51, 52, 53, 60, 69, 78]
>>> G = nx.sudoku_graph(2)
>>> G.number_of_nodes()
16
>>> G.number_of_edges()
56
```

CHAPTER

SIX

LINEAR ALGEBRA

6.1 Graph Matrix

Adjacency matrix and incidence matrix of graphs.

adjacency_matrix(G[, nodelist, dtype, weight])	Returns adjacency matrix of G.
incidence_matrix(G[, nodelist, edgelist, ...])	Returns incidence matrix of G.

6.1.1 adjacency_matrix

adjacency_matrix(*G*, *nodelist=None*, *dtype=None*, *weight='weight'*)

Returns adjacency matrix of G.

Parameters

G
[graph] A NetworkX graph

nodelist
[list, optional] The rows and columns are ordered according to the nodes in nodelist. If nodelist is None, then the ordering is produced by G.nodes().

dtype
[NumPy data-type, optional] The desired data-type for the array. If None, then the NumPy default is used.

weight
[string or None, optional (default='weight')] The edge data key used to provide each value in the matrix. If None, then each edge has weight 1.

Returns

A
[SciPy sparse array] Adjacency matrix representation of G.

See also:

to_numpy_array
to_scipy_sparse_array
to_dict_of_dicts
adjacency_spectrum

Notes

For directed graphs, entry i,j corresponds to an edge from i to j.

If you want a pure Python adjacency matrix representation try networkx.convert.to_dict_of_dicts which will return a dictionary-of-dictionaries format that can be addressed as a sparse matrix.

For MultiGraph/MultiDiGraph with parallel edges the weights are summed. See `to_numpy_array` for other options.

The convention used for self-loop edges in graphs is to assign the diagonal matrix entry value to the edge weight attribute (or the number 1 if the edge has no weight attribute). If the alternate convention of doubling the edge weight is desired the resulting SciPy sparse array can be modified as follows:

```
>>> G = nx.Graph([(1, 1)])
>>> A = nx.adjacency_matrix(G)
>>> print(A.todense())
[[1]]
>>> A.setdiag(A.diagonal() * 2)
>>> print(A.todense())
[[2]]
```

6.1.2 incidence_matrix

incidence_matrix(*G*, *nodelist=None*, *edgelist=None*, *oriented=False*, *weight=None*)

Returns incidence matrix of G.

The incidence matrix assigns each row to a node and each column to an edge. For a standard incidence matrix a 1 appears wherever a row's node is incident on the column's edge. For an oriented incidence matrix each edge is assigned an orientation (arbitrarily for undirected and aligning to direction for directed). A -1 appears for the source (tail) of an edge and 1 for the destination (head) of the edge. The elements are zero otherwise.

Parameters

G
[graph] A NetworkX graph

nodelist
[list, optional (default= all nodes in G)] The rows are ordered according to the nodes in nodelist. If nodelist is None, then the ordering is produced by G.nodes().

edgelist
[list, optional (default= all edges in G)] The columns are ordered according to the edges in edgelist. If edgelist is None, then the ordering is produced by G.edges().

oriented: bool, optional (default=False)
If True, matrix elements are +1 or -1 for the head or tail node respectively of each edge. If False, +1 occurs at both nodes.

weight
[string or None, optional (default=None)] The edge data key used to provide each value in the matrix. If None, then each edge has weight 1. Edge weights, if used, should be positive so that the orientation can provide the sign.

Returns

A
[SciPy sparse array] The incidence matrix of G.

Notes

For MultiGraph/MultiDiGraph, the edges in edgelist should be (u,v,key) 3-tuples.

"Networks are the best discrete model for so many problems in applied mathematics" [1].

References

[1]

6.2 Laplacian Matrix

Laplacian matrix of graphs.

laplacian_matrix(G[, nodelist, weight])	Returns the Laplacian matrix of G.
normalized_laplacian_matrix(G[, nodelist, ...])	Returns the normalized Laplacian matrix of G.
directed_laplacian_matrix(G[, nodelist, ...])	Returns the directed Laplacian matrix of G.
directed_combinatorial_laplacian_matrix(G[, ...])	Return the directed combinatorial Laplacian matrix of G.

6.2.1 laplacian_matrix

laplacian_matrix(*G*, *nodelist=None*, *weight='weight'*)

Returns the Laplacian matrix of G.

The graph Laplacian is the matrix L = D - A, where A is the adjacency matrix and D is the diagonal matrix of node degrees.

Parameters

G
[graph] A NetworkX graph

nodelist
[list, optional] 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')] The edge data key used to compute each value in the matrix. If None, then each edge has weight 1.

Returns

L
[SciPy sparse array] The Laplacian matrix of G.

See also:

to_numpy_array
normalized_laplacian_matrix
laplacian_spectrum

Notes

For MultiGraph, the edges weights are summed.

Examples

For graphs with multiple connected components, L is permutation-similar to a block diagonal matrix where each block is the respective Laplacian matrix for each component.

```
>>> G = nx.Graph([(1, 2), (2, 3), (4, 5)])
>>> print(nx.laplacian_matrix(G).toarray())
[[ 1 -1  0  0  0]
 [-1  2 -1  0  0]
 [ 0 -1  1  0  0]
 [ 0  0  0  1 -1]
 [ 0  0  0 -1  1]]
```

6.2.2 normalized_laplacian_matrix

normalized_laplacian_matrix(*G*, *nodelist=None*, *weight='weight'*)

Returns the normalized Laplacian matrix of G.

The normalized graph Laplacian is the matrix

$$N = D^{-1/2} L D^{-1/2}$$

where `L` is the graph Laplacian and `D` is the diagonal matrix of node degrees [1].

Parameters

G
[graph] A NetworkX graph

nodelist
[list, optional] 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')] The edge data key used to compute each value in the matrix. If None, then each edge has weight 1.

Returns

N
[SciPy sparse array] The normalized Laplacian matrix of G.

See also:

laplacian_matrix
normalized_laplacian_spectrum

Notes

For MultiGraph, the edges weights are summed. See `to_numpy_array()` for other options.

If the Graph contains selfloops, D is defined as `diag(sum(A, 1))`, where A is the adjacency matrix [2].

References

[1], [2]

6.2.3 directed_laplacian_matrix

directed_laplacian_matrix(*G*, *nodelist=None*, *weight='weight'*, *walk_type=None*, *alpha=0.95*)

Returns the directed Laplacian matrix of G.

The graph directed Laplacian is the matrix

$$L = I - (\Phi^{1/2} P \Phi^{-1/2} + \Phi^{-1/2} P^T \Phi^{1/2})/2$$

where `I` is the identity matrix, `P` is the transition matrix of the graph, and `Phi` a matrix with the Perron vector of `P` in the diagonal and zeros elsewhere [1].

Depending on the value of walk_type, `P` can be the transition matrix induced by a random walk, a lazy random walk, or a random walk with teleportation (PageRank).

Parameters

G
[DiGraph] A NetworkX graph

nodelist
[list, optional] 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')] 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)] If None, `P` is selected depending on the properties of the graph. Otherwise is one of 'random', 'lazy', or 'pagerank'

alpha
[real] (1 - alpha) is the teleportation probability used with pagerank

Returns

L
[NumPy matrix] Normalized Laplacian of G.

See also:

laplacian_matrix

Notes

Only implemented for DiGraphs

References

[1]

6.2.4 directed_combinatorial_laplacian_matrix

directed_combinatorial_laplacian_matrix(*G*, *nodelist=None*, *weight='weight'*, *walk_type=None*, *alpha=0.95*)

Return the directed combinatorial Laplacian matrix of G.

The graph directed combinatorial Laplacian is the matrix

$$L = \Phi - (\Phi P + P^T \Phi)/2$$

where `P` is the transition matrix of the graph and `Phi` a matrix with the Perron vector of `P` in the diagonal and zeros elsewhere [1].

Depending on the value of walk_type, `P` can be the transition matrix induced by a random walk, a lazy random walk, or a random walk with teleportation (PageRank).

Parameters

G
[DiGraph] A NetworkX graph

nodelist
[list, optional] 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')] 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)] If None, `P` is selected depending on the properties of the graph. Otherwise is one of 'random', 'lazy', or 'pagerank'

alpha
[real] (1 - alpha) is the teleportation probability used with pagerank

Returns

L
[NumPy matrix] Combinatorial Laplacian of G.

See also:

laplacian_matrix

Notes

Only implemented for DiGraphs

References

[1]

6.3 Bethe Hessian Matrix

Bethe Hessian or deformed Laplacian matrix of graphs.

bethe_hessian_matrix(G[, r, nodelist])	Returns the Bethe Hessian matrix of G.

6.3.1 bethe_hessian_matrix

bethe_hessian_matrix(*G*, *r=None*, *nodelist=None*)

Returns the Bethe Hessian matrix of G.

The Bethe Hessian is a family of matrices parametrized by r, defined as H(r) = (r^2 - 1) I - r A + D where A is the adjacency matrix, D is the diagonal matrix of node degrees, and I is the identify matrix. It is equal to the graph laplacian when the regularizer r = 1.

The default choice of regularizer should be the ratio [2]

$$r_m = \left(\sum k_i\right)^{-1} \left(\sum k_i^2\right) - 1$$

Parameters

G
[Graph] A NetworkX graph

r
[float] Regularizer parameter

nodelist
[list, optional] The rows and columns are ordered according to the nodes in nodelist. If nodelist is None, then the ordering is produced by `G.nodes()`.

Returns

H
[scipy.sparse.csr_array] The Bethe Hessian matrix of `G`, with parameter `r`.

See also:

bethe_hessian_spectrum
adjacency_matrix
laplacian_matrix

References

[1], [2]

Examples

```
>>> k = [3, 2, 2, 1, 0]
>>> G = nx.havel_hakimi_graph(k)
>>> H = nx.bethe_hessian_matrix(G)
>>> H.toarray()
array([[ 3.5625, -1.25  , -1.25  , -1.25  ,  0.    ],
       [-1.25  ,  2.5625, -1.25  ,  0.    ,  0.    ],
       [-1.25  , -1.25  ,  2.5625,  0.    ,  0.    ],
       [-1.25  ,  0.    ,  0.    ,  1.5625,  0.    ],
       [ 0.    ,  0.    ,  0.    ,  0.    ,  0.5625]])
```

6.4 Algebraic Connectivity

Algebraic connectivity and Fiedler vectors of undirected graphs.

algebraic_connectivity(G[, weight, ...])	Returns the algebraic connectivity of an undirected graph.
fiedler_vector(G[, weight, normalized, tol, ...])	Returns the Fiedler vector of a connected undirected graph.
spectral_ordering(G[, weight, normalized, ...])	Compute the spectral_ordering of a graph.

6.4.1 algebraic_connectivity

algebraic_connectivity(*G*, *weight='weight'*, *normalized=False*, *tol=1e-08*, *method='tracemin_pcg'*, *seed=None*)

Returns the algebraic connectivity of an undirected graph.

The algebraic connectivity of a connected undirected graph is the second smallest eigenvalue of its Laplacian matrix.

Parameters

G
[NetworkX graph] An undirected 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

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

algebraic_connectivity
[float] Algebraic connectivity.

Raises

NetworkXNotImplemented
If G is directed.

NetworkXError
If G has less than two nodes.

See also:

laplacian_matrix

Notes

Edge weights are interpreted by their absolute values. For MultiGraph's, weights of parallel edges are summed. Zero-weighted edges are ignored.

Examples

For undirected graphs algebraic connectivity can tell us if a graph is connected or not `G` is connected iff `algebraic_connectivity(G) > 0`:

```
>>> G = nx.complete_graph(5)
>>> nx.algebraic_connectivity(G) > 0
True
>>> G.add_node(10)  # G is no longer connected
>>> nx.algebraic_connectivity(G) > 0
False
```

6.4.2 fiedler_vector

fiedler_vector(*G*, *weight='weight'*, *normalized=False*, *tol=1e-08*, *method='tracemin_pcg'*, *seed=None*)

Returns the Fiedler vector of a connected undirected graph.

The Fiedler vector of a connected undirected graph is the eigenvector corresponding to the second smallest eigenvalue of the Laplacian matrix of the graph.

Parameters

G
[NetworkX graph] An undirected 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

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

fiedler_vector
[NumPy array of floats.] Fiedler vector.

Raises

NetworkXNotImplemented
If G is directed.

NetworkXError
If G has less than two nodes or is not connected.

See also:

laplacian_matrix

Notes

Edge weights are interpreted by their absolute values. For MultiGraph's, weights of parallel edges are summed. Zero-weighted edges are ignored.

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.

6.4.3 spectral_ordering

spectral_ordering(*G*, *weight='weight'*, *normalized=False*, *tol=1e-08*, *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

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

spectral_ordering
[NumPy array of floats.] Spectral ordering of nodes.

Raises

NetworkXError
If G is empty.

See also:

`laplacian_matrix`

Notes

Edge weights are interpreted by their absolute values. For MultiGraph's, weights of parallel edges are summed. Zero-weighted edges are ignored.

6.5 Attribute Matrices

Functions for constructing matrix-like objects from graph attributes.

attr_matrix(G[, edge_attr, node_attr, ...])	Returns the attribute matrix using attributes from `G` as a numpy array.
attr_sparse_matrix(G[, edge_attr, ...])	Returns a SciPy sparse array using attributes from G.

6.5.1 attr_matrix

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).

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.

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.

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)
```

(continues on next page)

(continued from previous page)

```
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.

```
>>> nx.attr_matrix(G, edge_attr="weight", node_attr="color", rc_order=rc)
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

6.5.2 attr_sparse_matrix

attr_sparse_matrix(*G*, *edge_attr=None*, *node_attr=None*, *normalized=False*, *rc_order=None*, *dtype=None*)

Returns a SciPy sparse array using attributes from G.

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 NumPy 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 matirx. 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).

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.

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.

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

6.6 Modularity Matrices

Modularity matrix of graphs.

modularity_matrix(G[, nodelist, weight])	Returns the modularity matrix of G.
directed_modularity_matrix(G[, nodelist, weight])	Returns the directed modularity matrix of G.

6.6.1 modularity_matrix

modularity_matrix(*G*, *nodelist=None*, *weight=None*)

Returns the modularity matrix of G.

The modularity matrix is the matrix B = A - <A>, where A is the adjacency matrix and <A> is the average adjacency matrix, assuming that the graph is described by the configuration model.

More specifically, the element B_ij of B is defined as

$$A_{ij} - \frac{k_i k_j}{2m}$$

where k_i is the degree of node i, and where m is the number of edges in the graph. When weight is set to a name of an attribute edge, Aij, k_i, k_j and m are computed using its value.

Parameters

G
[Graph] A NetworkX graph

nodelist
[list, optional] 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=None)] The edge attribute that holds the numerical value used for the edge weight. If None then all edge weights are 1.

Returns

B
[Numpy array] The modularity matrix of G.

See also:

to_numpy_array
modularity_spectrum
adjacency_matrix
directed_modularity_matrix

References

[1]

Examples

```
>>> k = [3, 2, 2, 1, 0]
>>> G = nx.havel_hakimi_graph(k)
>>> B = nx.modularity_matrix(G)
```

6.6.2 directed_modularity_matrix

directed_modularity_matrix(*G*, *nodelist=None*, *weight=None*)

Returns the directed modularity matrix of G.

The modularity matrix is the matrix B = A - <A>, where A is the adjacency matrix and <A> is the expected adjacency matrix, assuming that the graph is described by the configuration model.

More specifically, the element B_ij of B is defined as

$$B_{ij} = A_{ij} - k_i^{out} k_j^{in} / m$$

where k_i^{in} is the in degree of node i, and k_j^{out} is the out degree of node j, with m the number of edges in the graph. When weight is set to a name of an attribute edge, Aij, k_i, k_j and m are computed using its value.

Parameters

G
: [DiGraph] A NetworkX DiGraph

nodelist
: [list, optional] 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=None)] The edge attribute that holds the numerical value used for the edge weight. If None then all edge weights are 1.

Returns

B
: [Numpy array] The modularity matrix of G.

See also:

to_numpy_array
modularity_spectrum
adjacency_matrix
modularity_matrix

Notes

NetworkX defines the element A_ij of the adjacency matrix as 1 if there is a link going from node i to node j. Leicht and Newman use the opposite definition. This explains the different expression for B_ij.

References

[1]

Examples

```
>>> G = nx.DiGraph()
>>> G.add_edges_from(
...     (
...         (1, 2),
...         (1, 3),
...         (3, 1),
...         (3, 2),
...         (3, 5),
...         (4, 5),
...         (4, 6),
...         (5, 4),
...         (5, 6),
...         (6, 4),
...     )
... )
>>> B = nx.directed_modularity_matrix(G)
```

6.7 Spectrum

Eigenvalue spectrum of graphs.

adjacency_spectrum(G[, weight])	Returns eigenvalues of the adjacency matrix of G.
laplacian_spectrum(G[, weight])	Returns eigenvalues of the Laplacian of G
bethe_hessian_spectrum(G[, r])	Returns eigenvalues of the Bethe Hessian matrix of G.
normalized_laplacian_spectrum(G[, weight])	Return eigenvalues of the normalized Laplacian of G
modularity_spectrum(G)	Returns eigenvalues of the modularity matrix of G.

6.7.1 adjacency_spectrum

adjacency_spectrum(*G*, *weight='weight'*)

Returns eigenvalues of the adjacency matrix of G.

Parameters

G

[graph] A NetworkX graph

weight
[string or None, optional (default='weight')] The edge data key used to compute each value in the matrix. If None, then each edge has weight 1.

Returns

evals
[NumPy array] Eigenvalues

See also:

`adjacency_matrix`

Notes

For MultiGraph/MultiDiGraph, the edges weights are summed. See to_numpy_array for other options.

6.7.2 laplacian_spectrum

laplacian_spectrum(*G*, *weight='weight'*)

Returns eigenvalues of the Laplacian of G

Parameters

G
[graph] A NetworkX graph

weight
[string or None, optional (default='weight')] The edge data key used to compute each value in the matrix. If None, then each edge has weight 1.

Returns

evals
[NumPy array] Eigenvalues

See also:

`laplacian_matrix`

Notes

For MultiGraph/MultiDiGraph, the edges weights are summed. See *to_numpy_array()* for other options.

Examples

The multiplicity of 0 as an eigenvalue of the laplacian matrix is equal to the number of connected components of G.

```
>>> G = nx.Graph()  # Create a graph with 5 nodes and 3 connected components
>>> G.add_nodes_from(range(5))
>>> G.add_edges_from([(0, 2), (3, 4)])
>>> nx.laplacian_spectrum(G)
array([0., 0., 0., 2., 2.])
```

6.7.3 bethe_hessian_spectrum

bethe_hessian_spectrum(*G*, *r=None*)

Returns eigenvalues of the Bethe Hessian matrix of G.

Parameters

G
[Graph] A NetworkX Graph or DiGraph

r
[float] Regularizer parameter

Returns

evals
[NumPy array] Eigenvalues

See also:

bethe_hessian_matrix

References

[1]

6.7.4 normalized_laplacian_spectrum

normalized_laplacian_spectrum(*G*, *weight='weight'*)

Return eigenvalues of the normalized Laplacian of G

Parameters

G
[graph] A NetworkX graph

weight
[string or None, optional (default='weight')] The edge data key used to compute each value in the matrix. If None, then each edge has weight 1.

Returns

evals
[NumPy array] Eigenvalues

See also:

normalized_laplacian_matrix

Notes

For MultiGraph/MultiDiGraph, the edges weights are summed. See to_numpy_array for other options.

6.7.5 modularity_spectrum

modularity_spectrum(*G*)

Returns eigenvalues of the modularity matrix of G.

Parameters

G
[Graph] A NetworkX Graph or DiGraph

Returns

evals
[NumPy array] Eigenvalues

See also:

modularity_matrix

References

[1]

CHAPTER

SEVEN

CONVERTING TO AND FROM OTHER DATA FORMATS

7.1 To NetworkX Graph

Functions to convert NetworkX graphs to and from other formats.

The preferred way of converting data to a NetworkX graph is through the graph constructor. The constructor calls the to_networkx_graph() function which attempts to guess the input type and convert it automatically.

7.1.1 Examples

Create a graph with a single edge from a dictionary of dictionaries

```
>>> d = {0: {1: 1}}  # dict-of-dicts single edge (0,1)
>>> G = nx.Graph(d)
```

7.1.2 See Also

nx_agraph, nx_pydot

to_networkx_graph(data[, create_using, ...])	Make a NetworkX graph from a known data structure.

7.1.3 to_networkx_graph

to_networkx_graph(*data*, *create_using=None*, *multigraph_input=False*)

Make a NetworkX graph from a known data structure.

The preferred way to call this is automatically from the class constructor

```
>>> d = {0: {1: {"weight": 1}}}  # dict-of-dicts single edge (0,1)
>>> G = nx.Graph(d)
```

instead of the equivalent

```
>>> G = nx.from_dict_of_dicts(d)
```

Parameters

data
[object to be converted]

Current known types are:
any NetworkX graph dict-of-dicts dict-of-lists container (e.g. set, list, tuple) of edges iterator (e.g. itertools.chain) that produces edges generator of edges Pandas DataFrame (row per edge) 2D numpy array scipy sparse array pygraphviz agraph

create_using
[NetworkX graph constructor, optional (default=nx.Graph)] Graph type to create. If graph instance, then cleared before populated.

multigraph_input
[bool (default False)] If True and data is a dict_of_dicts, try to create a multigraph assuming dict_of_dict_of_lists. If data and create_using are both multigraphs then create a multigraph from a multigraph.

7.2 Dictionaries

to_dict_of_dicts(G[, nodelist, edge_data])	Returns adjacency representation of graph as a dictionary of dictionaries.
from_dict_of_dicts(d[, create_using, ...])	Returns a graph from a dictionary of dictionaries.

7.2.1 to_dict_of_dicts

to_dict_of_dicts(*G*, *nodelist=None*, *edge_data=None*)

Returns adjacency representation of graph as a dictionary of dictionaries.

Parameters

G
[graph] A NetworkX graph

nodelist
[list] Use only nodes specified in nodelist

edge_data
[scalar, optional] If provided, the value of the dictionary will be set to `edge_data` for all edges. Usual values could be `1` or `True`. If `edge_data` is `None` (the default), the edge-data in `G` is used, resulting in a dict-of-dict-of-dicts. If `G` is a MultiGraph, the result will be a dict-of-dict-of-dict-of-dicts. See Notes for an approach to customize handling edge data. `edge_data` should *not* be a container.

Returns

dod
[dict] A nested dictionary representation of `G`. Note that the level of nesting depends on the type of `G` and the value of `edge_data` (see Examples).

See also:

from_dict_of_dicts, ***to_dict_of_lists***

Notes

For a more custom approach to handling edge data, try:

```
dod = {
    n: {
        nbr: custom(n, nbr, dd) for nbr, dd in nbrdict.items()
    }
    for n, nbrdict in G.adj.items()
}
```

where `custom` returns the desired edge data for each edge between `n` and `nbr`, given existing edge data `dd`.

Examples

```
>>> G = nx.path_graph(3)
>>> nx.to_dict_of_dicts(G)
{0: {1: {}}, 1: {0: {}, 2: {}}, 2: {1: {}}}
```

Edge data is preserved by default (`edge_data=None`), resulting in dict-of-dict-of-dicts where the innermost dictionary contains the edge data:

```
>>> G = nx.Graph()
>>> G.add_edges_from(
...     [
...         (0, 1, {'weight': 1.0}),
...         (1, 2, {'weight': 2.0}),
...         (2, 0, {'weight': 1.0}),
...     ]
... )
>>> d = nx.to_dict_of_dicts(G)
>>> d
{0: {1: {'weight': 1.0}, 2: {'weight': 1.0}},
 1: {0: {'weight': 1.0}, 2: {'weight': 2.0}},
 2: {1: {'weight': 2.0}, 0: {'weight': 1.0}}}
>>> d[1][2]['weight']
2.0
```

If `edge_data` is not `None`, edge data in the original graph (if any) is replaced:

```
>>> d = nx.to_dict_of_dicts(G, edge_data=1)
>>> d
{0: {1: 1, 2: 1}, 1: {0: 1, 2: 1}, 2: {1: 1, 0: 1}}
>>> d[1][2]
1
```

This also applies to MultiGraphs: edge data is preserved by default:

```
>>> G = nx.MultiGraph()
>>> G.add_edge(0, 1, key='a', weight=1.0)
'a'
>>> G.add_edge(0, 1, key='b', weight=5.0)
'b'
>>> d = nx.to_dict_of_dicts(G)
>>> d
{0: {1: {'a': {'weight': 1.0}, 'b': {'weight': 5.0}}},
```

(continues on next page)

(continued from previous page)

```
 1: {0: {'a': {'weight': 1.0}, 'b': {'weight': 5.0}}}}
>>> d[0][1]['b']['weight']
5.0
```

But multi edge data is lost if `edge_data` is not `None`:

```
>>> d = nx.to_dict_of_dicts(G, edge_data=10)
>>> d
{0: {1: 10}, 1: {0: 10}}
```

7.2.2 from_dict_of_dicts

from_dict_of_dicts(*d*, *create_using=None*, *multigraph_input=False*)

Returns a graph from a dictionary of dictionaries.

Parameters

d
: [dictionary of dictionaries] A dictionary of dictionaries adjacency representation.

create_using
: [NetworkX graph constructor, optional (default=nx.Graph)] Graph type to create. If graph instance, then cleared before populated.

multigraph_input
: [bool (default False)] When True, the dict `d` 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. Otherwise this routine assumes dict-of-dict-of-dict keyed by node to neighbor to edge data.

Examples

```
>>> dod = {0: {1: {"weight": 1}}}  # single edge (0,1)
>>> G = nx.from_dict_of_dicts(dod)
```

or

```
>>> G = nx.Graph(dod)  # use Graph constructor
```

7.3 Lists

to_dict_of_lists(G[, nodelist])	Returns adjacency representation of graph as a dictionary of lists.
from_dict_of_lists(d[, create_using])	Returns a graph from a dictionary of lists.
to_edgelist(G[, nodelist])	Returns a list of edges in the graph.
from_edgelist(edgelist[, create_using])	Returns a graph from a list of edges.

7.3.1 to_dict_of_lists

to_dict_of_lists(*G*, *nodelist=None*)

Returns adjacency representation of graph as a dictionary of lists.

Parameters

G
[graph] A NetworkX graph

nodelist
[list] Use only nodes specified in nodelist

Notes

Completely ignores edge data for MultiGraph and MultiDiGraph.

7.3.2 from_dict_of_lists

from_dict_of_lists(*d*, *create_using=None*)

Returns a graph from a dictionary of lists.

Parameters

d
[dictionary of lists] A dictionary of lists adjacency representation.

create_using
[NetworkX graph constructor, optional (default=nx.Graph)] Graph type to create. If graph instance, then cleared before populated.

Examples

```
>>> dol = {0: [1]}  # single edge (0,1)
>>> G = nx.from_dict_of_lists(dol)
```

or

```
>>> G = nx.Graph(dol)  # use Graph constructor
```

7.3.3 to_edgelist

to_edgelist(*G*, *nodelist=None*)

Returns a list of edges in the graph.

Parameters

G
[graph] A NetworkX graph

nodelist
[list] Use only nodes specified in nodelist

7.3.4 from_edgelist

from_edgelist(*edgelist*, *create_using=None*)

Returns a graph from a list of edges.

Parameters

edgelist
[list or iterator] Edge tuples

create_using
[NetworkX graph constructor, optional (default=nx.Graph)] Graph type to create. If graph instance, then cleared before populated.

Examples

```
>>> edgelist = [(0, 1)]  # single edge (0,1)
>>> G = nx.from_edgelist(edgelist)
```

or

```
>>> G = nx.Graph(edgelist)  # use Graph constructor
```

7.4 Numpy

Functions to convert NetworkX graphs to and from common data containers like numpy arrays, scipy sparse arrays, and pandas DataFrames.

The preferred way of converting data to a NetworkX graph is through the graph constructor. The constructor calls the *to_networkx_graph* function which attempts to guess the input type and convert it automatically.

7.4.1 Examples

Create a 10 node random graph from a numpy array

```
>>> import numpy as np
>>> rng = np.random.default_rng()
>>> a = rng.integers(low=0, high=2, size=(10, 10))
>>> DG = nx.from_numpy_array(a, create_using=nx.DiGraph)
```

or equivalently:

```
>>> DG = nx.DiGraph(a)
```

which calls *from_numpy_array* internally based on the type of `a`.

7.4.2 See Also

nx_agraph, nx_pydot

to_numpy_array(G[, nodelist, dtype, order, ...])	Returns the graph adjacency matrix as a NumPy array.
from_numpy_array(A[, parallel_edges, ...])	Returns a graph from a 2D NumPy array.

7.4.3 to_numpy_array

to_numpy_array(*G*, *nodelist=None*, *dtype=None*, *order=None*, *multigraph_weight=<built-in function sum>*, *weight='weight'*, *nonedge=0.0*)

Returns the graph adjacency matrix as a NumPy array.

Parameters

G
[graph] The NetworkX graph used to construct the NumPy array.

nodelist
[list, optional] The rows and columns are ordered according to the nodes in `nodelist`. If `nodelist` is `None`, then the ordering is produced by `G.nodes()`.

dtype
[NumPy data type, optional] A NumPy data type used to initialize the array. If None, then the NumPy default is used. The dtype can be structured if `weight=None`, in which case the dtype field names are used to look up edge attributes. The result is a structured array where each named field in the dtype corresponds to the adjaceny for that edge attribute. See examples for details.

order
[{'C', 'F'}, optional] Whether to store multidimensional data in C- or Fortran-contiguous (row- or column-wise) order in memory. If None, then the NumPy default is used.

multigraph_weight
[callable, optional] An function that determines how weights in multigraphs are handled. The function should accept a sequence of weights and return a single value. The default is to sum the weights of the multiple edges.

weight
[string or None optional (default = 'weight')] 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. `weight` must be `None` if a structured dtype is used.

nonedge
[array_like (default = 0.0)] The value used to represent non-edges in the adjaceny matrix. The array values corresponding to nonedges are typically set to zero. However, this could be undesirable if there are array 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

A
[NumPy ndarray] Graph adjacency matrix

Raises

NetworkXError
If `dtype` is a structured dtype and `G` is a multigraph

ValueError
If `dtype` is a structured dtype and `weight` is not `None`

See also:

`from_numpy_array`

Notes

For directed graphs, entry `i, j` corresponds to an edge from `i` to `j`.

Entries in the adjacency matrix are given by 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 adjacency 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 array 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 NumPy array can be modified as follows:

```
>>> import numpy as np
>>> G = nx.Graph([(1, 1)])
>>> A = nx.to_numpy_array(G)
>>> A
array([[1.]])
>>> A[np.diag_indices_from(A)] *= 2
>>> A
array([[2.]])
```

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_numpy_array(G, nodelist=[0, 1, 2])
array([[0., 2., 0.],
       [1., 0., 0.],
       [0., 0., 4.]])
```

When `nodelist` argument is used, nodes of `G` which do not appear in the `nodelist` and their edges are not included in the adjacency matrix. Here is an example:

```
>>> G = nx.Graph()
>>> G.add_edge(3, 1)
>>> G.add_edge(2, 0)
>>> G.add_edge(2, 1)
```

(continues on next page)

(continued from previous page)

```
>>> G.add_edge(3, 0)
>>> nx.to_numpy_array(G, nodelist=[1, 2, 3])
array([[0., 1., 1.],
       [1., 0., 0.],
       [1., 0., 0.]])
```

This function can also be used to create adjacency matrices for multiple edge attributes with structured dtypes:

```
>>> G = nx.Graph()
>>> G.add_edge(0, 1, weight=10)
>>> G.add_edge(1, 2, cost=5)
>>> G.add_edge(2, 3, weight=3, cost=-4.0)
>>> dtype = np.dtype([("weight", int), ("cost", float)])
>>> A = nx.to_numpy_array(G, dtype=dtype, weight=None)
>>> A["weight"]
array([[ 0, 10,  0,  0],
       [10,  0,  1,  0],
       [ 0,  1,  0,  3],
       [ 0,  0,  3,  0]])
>>> A["cost"]
array([[ 0.,  1.,  0.,  0.],
       [ 1.,  0.,  5.,  0.],
       [ 0.,  5.,  0., -4.],
       [ 0.,  0., -4.,  0.]])
```

As stated above, the argument “nonedge” is useful especially when there are actually edges with weight 0 in the graph. Setting a nonedge value different than 0, makes it much clearer to differentiate such 0-weighted edges and actual nonedge values.

```
>>> G = nx.Graph()
>>> G.add_edge(3, 1, weight=2)
>>> G.add_edge(2, 0, weight=0)
>>> G.add_edge(2, 1, weight=0)
>>> G.add_edge(3, 0, weight=1)
>>> nx.to_numpy_array(G, nonedge=-1.)
array([[-1.,  2., -1.,  1.],
       [ 2., -1.,  0., -1.],
       [-1.,  0., -1.,  0.],
       [ 1., -1.,  0., -1.]])
```

7.4.4 from_numpy_array

from_numpy_array(*A*, *parallel_edges=False*, *create_using=None*)

Returns a graph from a 2D NumPy array.

The 2D NumPy array is interpreted as an adjacency matrix for the graph.

Parameters

A

[a 2D numpy.ndarray] An adjacency matrix representation of a graph

parallel_edges

[Boolean] If this is True, `create_using` is a multigraph, and `A` is an integer array, then entry *(i, j)* in the array is interpreted as the number of parallel edges joining vertices *i* and *j* in

the graph. If it is False, then the entries in the array are interpreted as the weight of a single edge joining the vertices.

create_using
[NetworkX graph constructor, optional (default=nx.Graph)] Graph type to create. If graph instance, then cleared before populated.

See also:

to_numpy_array

Notes

For directed graphs, explicitly mention create_using=nx.DiGraph, and entry i,j of A corresponds to an edge from i to j.

If `create_using` is *networkx.MultiGraph* or *networkx.MultiDiGraph*, `parallel_edges` is True, and the entries of `A` are of type `int`, then this function returns a multigraph (of the same type as `create_using`) with parallel edges.

If `create_using` indicates an undirected multigraph, then only the edges indicated by the upper triangle of the array `A` will be added to the graph.

If the NumPy array has a single data type for each array entry it will be converted to an appropriate Python data type.

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.

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
```

7.5 Scipy

to_scipy_sparse_array(G[, nodelist, dtype, ...])	Returns the graph adjacency matrix as a SciPy sparse array.
from_scipy_sparse_array(A[, parallel_edges, ...])	Creates a new graph from an adjacency matrix given as a SciPy sparse array.

7.5.1 to_scipy_sparse_array

to_scipy_sparse_array(*G*, *nodelist=None*, *dtype=None*, *weight='weight'*, *format='csr'*)

Returns the graph adjacency matrix as a SciPy sparse array.

Parameters

G
[graph] The NetworkX graph used to construct the sparse matrix.

nodelist
[list, optional] The rows and columns are ordered according to the nodes in `nodelist`. If `nodelist` is None, then the ordering is produced by G.nodes().

dtype
[NumPy data-type, optional] A valid NumPy dtype used to initialize the array. If None, then the NumPy default is used.

weight
[string or None optional (default='weight')] The edge attribute that holds the numerical value used for the edge weight. If None then all edge weights are 1.

format
[str in {'bsr', 'csr', 'csc', 'coo', 'lil', 'dia', 'dok'}] The type of the matrix to be returned (default 'csr'). For some algorithms different implementations of sparse matrices can perform better. See [1] for details.

Returns

A
[SciPy sparse array] Graph adjacency matrix.

Notes

For directed graphs, matrix entry i,j corresponds to an edge from i to j.

The matrix entries are populated using the edge attribute held in parameter weight. When an edge does not have that attribute, the value of the entry is 1.

For multiple edges the matrix values are the sums of the edge weights.

When `nodelist` does not contain every node in `G`, the adjacency 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 SciPy sparse array can be modified as follows:

```
>>> G = nx.Graph([(1, 1)])
>>> A = nx.to_scipy_sparse_array(G)
>>> print(A.todense())
[[1]]
>>> A.setdiag(A.diagonal() * 2)
>>> print(A.toarray())
[[2]]
```

References

[1]

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
>>> S = nx.to_scipy_sparse_array(G, nodelist=[0, 1, 2])
>>> print(S.toarray())
[[0 2 0]
 [1 0 0]
 [0 0 4]]
```

7.5.2 from_scipy_sparse_array

from_scipy_sparse_array(*A*, *parallel_edges=False*, *create_using=None*, *edge_attribute='weight'*)

Creates a new graph from an adjacency matrix given as a SciPy sparse array.

Parameters

A: scipy.sparse array
An adjacency matrix representation of a graph

parallel_edges
[Boolean] If this is True, `create_using` is a multigraph, and `A` is an integer matrix, then entry *(i, j)* in the matrix is interpreted as the number of parallel edges joining vertices *i* and *j* in the graph. If it is False, then the entries in the matrix are interpreted as the weight of a single edge joining the vertices.

create_using
[NetworkX graph constructor, optional (default=nx.Graph)] Graph type to create. If graph instance, then cleared before populated.

edge_attribute: string
Name of edge attribute to store matrix numeric value. The data will have the same type as the matrix entry (int, float, (real,imag)).

Notes

For directed graphs, explicitly mention create_using=nx.DiGraph, and entry i,j of A corresponds to an edge from i to j.

If `create_using` is *networkx.MultiGraph* or *networkx.MultiDiGraph*, `parallel_edges` is True, and the entries of `A` are of type `int`, then this function returns a multigraph (constructed from `create_using`) with parallel edges. In this case, `edge_attribute` will be ignored.

If `create_using` indicates an undirected multigraph, then only the edges indicated by the upper triangle of the matrix `A` will be added to the graph.

Examples

```
>>> import scipy as sp
>>> import scipy.sparse  # call as sp.sparse
>>> A = sp.sparse.eye(2, 2, 1)
>>> G = nx.from_scipy_sparse_array(A)
```

If `create_using` indicates a multigraph and the matrix 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 = sp.sparse.csr_array([[1, 1], [1, 2]])
>>> G = nx.from_scipy_sparse_array(A, create_using=nx.MultiGraph)
>>> G[1][1]
AtlasView({0: {'weight': 2}})
```

If `create_using` indicates a multigraph and the matrix 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 = sp.sparse.csr_array([[1, 1], [1, 2]])
>>> G = nx.from_scipy_sparse_array(
...     A, parallel_edges=True, create_using=nx.MultiGraph
... )
>>> G[1][1]
AtlasView({0: {'weight': 1}, 1: {'weight': 1}})
```

7.6 Pandas

to_pandas_adjacency(G[, nodelist, dtype, ...])	Returns the graph adjacency matrix as a Pandas DataFrame.
from_pandas_adjacency(df[, create_using])	Returns a graph from Pandas DataFrame.
to_pandas_edgelist(G[, source, target, ...])	Returns the graph edge list as a Pandas DataFrame.
from_pandas_edgelist(df[, source, target, ...])	Returns a graph from Pandas DataFrame containing an edge list.

7.6.1 to_pandas_adjacency

to_pandas_adjacency(*G*, *nodelist=None*, *dtype=None*, *order=None*, *multigraph_weight=<built-in function sum>*, *weight='weight'*, *nonedge=0.0*)

Returns the graph adjacency matrix as a Pandas DataFrame.

Parameters

G
[graph] The NetworkX graph used to construct the Pandas DataFrame.

nodelist
[list, optional] 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
>>> pd.options.display.max_columns = 20
>>> import numpy as np
>>> G = nx.Graph([(1, 1)])
>>> df = nx.to_pandas_adjacency(G, dtype=int)
>>> df
   1
1  1
>>> df.values[np.diag_indices_from(df)] *= 2
>>> df
   1
1  2
```

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
```

7.6.2 from_pandas_adjacency

from_pandas_adjacency(*df*, *create_using=None*)

Returns a graph from Pandas DataFrame.

The Pandas DataFrame is interpreted as an adjacency matrix for the graph.

Parameters

df

[Pandas DataFrame] An adjacency matrix representation of a graph

create_using
: [NetworkX graph constructor, optional (default=nx.Graph)] Graph type to create. If graph instance, then cleared before populated.

See also:

to_pandas_adjacency

Notes

For directed graphs, explicitly mention create_using=nx.DiGraph, and entry i,j of df corresponds to an edge from i to j.

If `df` has a single data type for each entry it will be converted to an appropriate Python data type.

If `df` has a user-specified compound data type the names of the data fields will be used as attribute keys in the resulting NetworkX graph.

Examples

Simple integer weights on edges:

```
>>> import pandas as pd
>>> pd.options.display.max_columns = 20
>>> df = pd.DataFrame([[1, 1], [2, 1]])
>>> df
   0  1
0  1  1
1  2  1
>>> G = nx.from_pandas_adjacency(df)
>>> G.name = "Graph from pandas adjacency matrix"
>>> print(G)
Graph named 'Graph from pandas adjacency matrix' with 2 nodes and 3 edges
```

7.6.3 to_pandas_edgelist

to_pandas_edgelist(*G*, *source='source'*, *target='target'*, *nodelist=None*, *dtype=None*, *edge_key=None*)

Returns the graph edge list as a Pandas DataFrame.

Parameters

G
: [graph] The NetworkX graph used to construct the Pandas DataFrame.

source
: [str or int, optional] A valid column name (string or integer) for the source nodes (for the directed case).

target
: [str or int, optional] A valid column name (string or integer) for the target nodes (for the directed case).

nodelist
: [list, optional] Use only nodes specified in nodelist

dtype
[dtype, default None] Use to create the DataFrame. Data type to force. Only a single dtype is allowed. If None, infer.

edge_key
[str or int or None, optional (default=None)] A valid column name (string or integer) for the edge keys (for the multigraph case). If None, edge keys are not stored in the DataFrame.

Returns

df
[Pandas DataFrame] Graph edge list

Examples

```
>>> G = nx.Graph(
...     [
...         ("A", "B", {"cost": 1, "weight": 7}),
...         ("C", "E", {"cost": 9, "weight": 10}),
...     ]
... )
>>> df = nx.to_pandas_edgelist(G, nodelist=["A", "C"])
>>> df[["source", "target", "cost", "weight"]]
  source target  cost  weight
0      A      B     1       7
1      C      E     9      10
```

```
>>> G = nx.MultiGraph([('A', 'B', {'cost': 1}), ('A', 'B', {'cost': 9})])
>>> df = nx.to_pandas_edgelist(G, nodelist=['A', 'C'], edge_key='ekey')
>>> df[['source', 'target', 'cost', 'ekey']]
  source target  cost  ekey
0      A      B     1     0
1      A      B     9     1
```

7.6.4 from_pandas_edgelist

from_pandas_edgelist(*df*, *source='source'*, *target='target'*, *edge_attr=None*, *create_using=None*, *edge_key=None*)

Returns a graph from Pandas DataFrame containing an edge list.

The Pandas DataFrame should contain at least two columns of node names and zero or more columns of edge attributes. Each row will be processed as one edge instance.

Note: This function iterates over DataFrame.values, which is not guaranteed to retain the data type across columns in the row. This is only a problem if your row is entirely numeric and a mix of ints and floats. In that case, all values will be returned as floats. See the DataFrame.iterrows documentation for an example.

Parameters

df
[Pandas DataFrame] An edge list representation of a graph

source
[str or int] A valid column name (string or integer) for the source nodes (for the directed case).

target
[str or int] A valid column name (string or integer) for the target nodes (for the directed case).

edge_attr
: [str or int, iterable, True, or None] A valid column name (str or int) or iterable of column names that are used to retrieve items and add them to the graph as edge attributes. If `True`, all of the remaining columns will be added. If `None`, no edge attributes are added to the graph.

create_using
: [NetworkX graph constructor, optional (default=nx.Graph)] Graph type to create. If graph instance, then cleared before populated.

edge_key
: [str or None, optional (default=None)] A valid column name for the edge keys (for a MultiGraph). The values in this column are used for the edge keys when adding edges if create_using is a multigraph.

See also:

to_pandas_edgelist

Examples

Simple integer weights on edges:

```
>>> import pandas as pd
>>> pd.options.display.max_columns = 20
>>> import numpy as np
>>> rng = np.random.RandomState(seed=5)
>>> ints = rng.randint(1, 11, size=(3, 2))
>>> a = ["A", "B", "C"]
>>> b = ["D", "A", "E"]
>>> df = pd.DataFrame(ints, columns=["weight", "cost"])
>>> df[0] = a
>>> df["b"] = b
>>> df[["weight", "cost", 0, "b"]]
   weight  cost  0  b
0       4     7  A  D
1       7     1  B  A
2      10     9  C  E
>>> G = nx.from_pandas_edgelist(df, 0, "b", ["weight", "cost"])
>>> G["E"]["C"]["weight"]
10
>>> G["E"]["C"]["cost"]
9
>>> edges = pd.DataFrame(
...     {
...         "source": [0, 1, 2],
...         "target": [2, 2, 3],
...         "weight": [3, 4, 5],
...         "color": ["red", "blue", "blue"],
...     }
... )
>>> G = nx.from_pandas_edgelist(edges, edge_attr=True)
>>> G[0][2]["color"]
'red'
```

Build multigraph with custom keys:

```
>>> edges = pd.DataFrame(
...     {
...         "source": [0, 1, 2, 0],
...         "target": [2, 2, 3, 2],
...         "my_edge_key": ["A", "B", "C", "D"],
...         "weight": [3, 4, 5, 6],
...         "color": ["red", "blue", "blue", "blue"],
...     }
... )
>>> G = nx.from_pandas_edgelist(
...     edges,
...     edge_key="my_edge_key",
...     edge_attr=["weight", "color"],
...     create_using=nx.MultiGraph(),
... )
>>> G[0][2]
AtlasView({'A': {'weight': 3, 'color': 'red'}, 'D': {'weight': 6, 'color': 'blue'}
↪})
```

CHAPTER

EIGHT

RELABELING NODES

8.1 Relabeling

convert_node_labels_to_integers(G[, ...])	Returns a copy of the graph G with the nodes relabeled using consecutive integers.
relabel_nodes(G, mapping[, copy])	Relabel the nodes of the graph G according to a given mapping.

8.1.1 convert_node_labels_to_integers

convert_node_labels_to_integers(*G*, *first_label=0*, *ordering='default'*, *label_attribute=None*)

Returns a copy of the graph G with the nodes relabeled using consecutive integers.

Parameters

G
[graph] A NetworkX graph

first_label
[int, optional (default=0)] An integer specifying the starting offset in numbering nodes. The new integer labels are numbered first_label, …, n-1+first_label.

ordering
[string] “default” : inherit node ordering from G.nodes() “sorted” : inherit node ordering from sorted(G.nodes()) “increasing degree” : nodes are sorted by increasing degree “decreasing degree” : nodes are sorted by decreasing degree

label_attribute
[string, optional (default=None)] Name of node attribute to store old label. If None no attribute is created.

See also:

relabel_nodes

Notes

Node and edge attribute data are copied to the new (relabeled) graph.

There is no guarantee that the relabeling of nodes to integers will give the same two integers for two (even identical graphs). Use the `ordering` argument to try to preserve the order.

8.1.2 relabel_nodes

relabel_nodes(*G*, *mapping*, *copy=True*)

Relabel the nodes of the graph G according to a given mapping.

The original node ordering may not be preserved if `copy` is `False` and the mapping includes overlap between old and new labels.

Parameters

G
[graph] A NetworkX graph

mapping
[dictionary] A dictionary with the old labels as keys and new labels as values. A partial mapping is allowed. Mapping 2 nodes to a single node is allowed. Any non-node keys in the mapping are ignored.

copy
[bool (optional, default=True)] If True return a copy, or if False relabel the nodes in place.

See also:

convert_node_labels_to_integers

Notes

Only the nodes specified in the mapping will be relabeled. Any non-node keys in the mapping are ignored.

The keyword setting copy=False modifies the graph in place. Relabel_nodes avoids naming collisions by building a directed graph from `mapping` which specifies the order of relabelings. Naming collisions, such as a->b, b->c, are ordered such that "b" gets renamed to "c" before "a" gets renamed "b". In cases of circular mappings (e.g. a->b, b->a), modifying the graph is not possible in-place and an exception is raised. In that case, use copy=True.

If a relabel operation on a multigraph would cause two or more edges to have the same source, target and key, the second edge must be assigned a new key to retain all edges. The new key is set to the lowest non-negative integer not already used as a key for edges between these two nodes. Note that this means non-numeric keys may be replaced by numeric keys.

Examples

To create a new graph with nodes relabeled according to a given dictionary:

```
>>> G = nx.path_graph(3)
>>> sorted(G)
[0, 1, 2]
>>> mapping = {0: "a", 1: "b", 2: "c"}
>>> H = nx.relabel_nodes(G, mapping)
>>> sorted(H)
['a', 'b', 'c']
```

Nodes can be relabeled with any hashable object, including numbers and strings:

```
>>> import string
>>> G = nx.path_graph(26)  # nodes are integers 0 through 25
>>> sorted(G)[:3]
[0, 1, 2]
>>> mapping = dict(zip(G, string.ascii_lowercase))
>>> G = nx.relabel_nodes(G, mapping)  # nodes are characters a through z
>>> sorted(G)[:3]
['a', 'b', 'c']
>>> mapping = dict(zip(G, range(1, 27)))
>>> G = nx.relabel_nodes(G, mapping)  # nodes are integers 1 through 26
>>> sorted(G)[:3]
[1, 2, 3]
```

To perform a partial in-place relabeling, provide a dictionary mapping only a subset of the nodes, and set the `copy` keyword argument to False:

```
>>> G = nx.path_graph(3)  # nodes 0-1-2
>>> mapping = {0: "a", 1: "b"}  # 0->'a' and 1->'b'
>>> G = nx.relabel_nodes(G, mapping, copy=False)
>>> sorted(G, key=str)
[2, 'a', 'b']
```

A mapping can also be given as a function:

```
>>> G = nx.path_graph(3)
>>> H = nx.relabel_nodes(G, lambda x: x ** 2)
>>> list(H)
[0, 1, 4]
```

In a multigraph, relabeling two or more nodes to the same new node will retain all edges, but may change the edge keys in the process:

```
>>> G = nx.MultiGraph()
>>> G.add_edge(0, 1, value="a")  # returns the key for this edge
0
>>> G.add_edge(0, 2, value="b")
0
>>> G.add_edge(0, 3, value="c")
0
>>> mapping = {1: 4, 2: 4, 3: 4}
>>> H = nx.relabel_nodes(G, mapping, copy=True)
>>> print(H[0])
{4: {0: {'value': 'a'}, 1: {'value': 'b'}, 2: {'value': 'c'}}}
```

This works for in-place relabeling too:

```
>>> G = nx.relabel_nodes(G, mapping, copy=False)
>>> print(G[0])
{4: {0: {'value': 'a'}, 1: {'value': 'b'}, 2: {'value': 'c'}}}
```

CHAPTER

NINE

READING AND WRITING GRAPHS

9.1 Adjacency List

Read and write NetworkX graphs as adjacency lists.

Adjacency list format is useful for graphs without data associated with nodes or edges and for nodes that can be meaningfully represented as strings.

9.1.1 Format

The adjacency list format consists of lines with node labels. The first label in a line is the source node. Further labels in the line are considered target nodes and are added to the graph along with an edge between the source node and target node.

The graph with edges a-b, a-c, d-e can be represented as the following adjacency list (anything following the # in a line is a comment):

```
a b c # source target target
d e
```

read_adjlist(path[, comments, delimiter, ...])	Read graph in adjacency list format from path.
write_adjlist(G, path[, comments, ...])	Write graph G in single-line adjacency-list format to path.
parse_adjlist(lines[, comments, delimiter, ...])	Parse lines of a graph adjacency list representation.
generate_adjlist(G[, delimiter])	Generate a single line of the graph G in adjacency list format.

9.1.2 read_adjlist

read_adjlist(*path*, *comments='#'*, *delimiter=None*, *create_using=None*, *nodetype=None*, *encoding='utf-8'*)

Read graph in adjacency list format from path.

Parameters

path
[string or file] Filename or file handle to read. Filenames ending in .gz or .bz2 will be uncompressed.

create_using
[NetworkX graph constructor, optional (default=nx.Graph)] Graph type to create. If graph instance, then cleared before populated.

nodetype
[Python type, optional] Convert nodes to this type.

comments
[string, optional] Marker for comment lines

delimiter
[string, optional] Separator for node labels. The default is whitespace.

Returns

G: NetworkX graph
The graph corresponding to the lines in adjacency list format.

See also:

write_adjlist

Notes

This format does not store graph or node data.

Examples

```
>>> G = nx.path_graph(4)
>>> nx.write_adjlist(G, "test.adjlist")
>>> G = nx.read_adjlist("test.adjlist")
```

The path can be a filehandle or a string with the name of the file. If a filehandle is provided, it has to be opened in 'rb' mode.

```
>>> fh = open("test.adjlist", "rb")
>>> G = nx.read_adjlist(fh)
```

Filenames ending in .gz or .bz2 will be compressed.

```
>>> nx.write_adjlist(G, "test.adjlist.gz")
>>> G = nx.read_adjlist("test.adjlist.gz")
```

The optional nodetype is a function to convert node strings to nodetype.

For example

```
>>> G = nx.read_adjlist("test.adjlist", nodetype=int)
```

will attempt to convert all nodes to integer type.

Since nodes must be hashable, the function nodetype must return hashable types (e.g. int, float, str, frozenset - or tuples of those, etc.)

The optional create_using parameter indicates the type of NetworkX graph created. The default is `nx.Graph`, an undirected graph. To read the data as a directed graph use

```
>>> G = nx.read_adjlist("test.adjlist", create_using=nx.DiGraph)
```

9.1.3 write_adjlist

write_adjlist(*G*, *path*, *comments='#'*, *delimiter=' '*, *encoding='utf-8'*)

Write graph G in single-line adjacency-list format to path.

Parameters

G
[NetworkX graph]

path
[string or file] Filename or file handle for data output. Filenames ending in .gz or .bz2 will be compressed.

comments
[string, optional] Marker for comment lines

delimiter
[string, optional] Separator for node labels

encoding
[string, optional] Text encoding.

See also:

read_adjlist, *generate_adjlist*

Notes

The default `delimiter=" "` will result in unexpected results if node names contain whitespace characters. To avoid this problem, specify an alternate delimiter when spaces are valid in node names. NB: This option is not available for data that isn't user-generated.

This format does not store graph, node, or edge data.

Examples

```
>>> G = nx.path_graph(4)
>>> nx.write_adjlist(G, "test.adjlist")
```

The path can be a filehandle or a string with the name of the file. If a filehandle is provided, it has to be opened in 'wb' mode.

```
>>> fh = open("test.adjlist", "wb")
>>> nx.write_adjlist(G, fh)
```

9.1.4 parse_adjlist

parse_adjlist(*lines*, *comments='#'*, *delimiter=None*, *create_using=None*, *nodetype=None*)

Parse lines of a graph adjacency list representation.

Parameters

lines
[list or iterator of strings] Input data in adjlist format

create_using
[NetworkX graph constructor, optional (default=nx.Graph)] Graph type to create. If graph instance, then cleared before populated.

nodetype
[Python type, optional] Convert nodes to this type.

comments
[string, optional] Marker for comment lines

delimiter
[string, optional] Separator for node labels. The default is whitespace.

Returns

G: NetworkX graph
The graph corresponding to the lines in adjacency list format.

See also:

read_adjlist

Examples

```
>>> lines = ["1 2 5", "2 3 4", "3 5", "4", "5"]
>>> G = nx.parse_adjlist(lines, nodetype=int)
>>> nodes = [1, 2, 3, 4, 5]
>>> all(node in G for node in nodes)
True
>>> edges = [(1, 2), (1, 5), (2, 3), (2, 4), (3, 5)]
>>> all((u, v) in G.edges() or (v, u) in G.edges() for (u, v) in edges)
True
```

9.1.5 generate_adjlist

generate_adjlist(*G*, *delimiter=' '*)

Generate a single line of the graph G in adjacency list format.

Parameters

G
[NetworkX graph]

delimiter
[string, optional] Separator for node labels

Returns

lines
[string] Lines of data in adjlist format.

See also:

write_adjlist, *read_adjlist*

Notes

The default `delimiter=" "` will result in unexpected results if node names contain whitespace characters. To avoid this problem, specify an alternate delimiter when spaces are valid in node names.

NB: This option is not available for data that isn't user-generated.

Examples

```
>>> G = nx.lollipop_graph(4, 3)
>>> for line in nx.generate_adjlist(G):
...     print(line)
0 1 2 3
1 2 3
2 3
3 4
4 5
5 6
6
```

9.2 Multiline Adjacency List

Read and write NetworkX graphs as multi-line adjacency lists.

The multi-line adjacency list format is useful for graphs with nodes that can be meaningfully represented as strings. With this format simple edge data can be stored but node or graph data is not.

9.2.1 Format

The first label in a line is the source node label followed by the node degree d. The next d lines are target node labels and optional edge data. That pattern repeats for all nodes in the graph.

The graph with edges a-b, a-c, d-e can be represented as the following adjacency list (anything following the # in a line is a comment):

```
# example.multiline-adjlist
a 2
b
c
d 1
e
```

read_multiline_adjlist(path[, comments, ...])	Read graph in multi-line adjacency list format from path.
write_multiline_adjlist(G, path[, ...])	Write the graph G in multiline adjacency list format to path
parse_multiline_adjlist(lines[, comments, ...])	Parse lines of a multiline adjacency list representation of a graph.
generate_multiline_adjlist(G[, delimiter])	Generate a single line of the graph G in multiline adjacency list format.

9.2.2 read_multiline_adjlist

read_multiline_adjlist(*path*, *comments='#'*, *delimiter=None*, *create_using=None*, *nodetype=None*, *edgetype=None*, *encoding='utf-8'*)

Read graph in multi-line adjacency list format from path.

Parameters

path
[string or file] Filename or file handle to read. Filenames ending in .gz or .bz2 will be uncompressed.

create_using
[NetworkX graph constructor, optional (default=nx.Graph)] Graph type to create. If graph instance, then cleared before populated.

nodetype
[Python type, optional] Convert nodes to this type.

edgetype
[Python type, optional] Convert edge data to this type.

comments
[string, optional] Marker for comment lines

delimiter
[string, optional] Separator for node labels. The default is whitespace.

Returns

G: NetworkX graph

See also:

write_multiline_adjlist

Notes

This format does not store graph, node, or edge data.

Examples

```
>>> G = nx.path_graph(4)
>>> nx.write_multiline_adjlist(G, "test.adjlist")
>>> G = nx.read_multiline_adjlist("test.adjlist")
```

The path can be a file or a string with the name of the file. If a file s provided, it has to be opened in 'rb' mode.

```
>>> fh = open("test.adjlist", "rb")
>>> G = nx.read_multiline_adjlist(fh)
```

Filenames ending in .gz or .bz2 will be compressed.

```
>>> nx.write_multiline_adjlist(G, "test.adjlist.gz")
>>> G = nx.read_multiline_adjlist("test.adjlist.gz")
```

The optional nodetype is a function to convert node strings to nodetype.

For example

```
>>> G = nx.read_multiline_adjlist("test.adjlist", nodetype=int)
```

will attempt to convert all nodes to integer type.

The optional edgetype is a function to convert edge data strings to edgetype.

```
>>> G = nx.read_multiline_adjlist("test.adjlist")
```

The optional create_using parameter is a NetworkX graph container. The default is Graph(), an undirected graph. To read the data as a directed graph use

```
>>> G = nx.read_multiline_adjlist("test.adjlist", create_using=nx.DiGraph)
```

9.2.3 write_multiline_adjlist

write_multiline_adjlist(*G*, *path*, *delimiter=' '*, *comments='#'*, *encoding='utf-8'*)

Write the graph G in multiline adjacency list format to path

Parameters

G
[NetworkX graph]

path
[string or file] Filename or file handle to write to. Filenames ending in .gz or .bz2 will be compressed.

comments
[string, optional] Marker for comment lines

delimiter
[string, optional] Separator for node labels

encoding
[string, optional] Text encoding.

See also:

read_multiline_adjlist

Examples

```
>>> G = nx.path_graph(4)
>>> nx.write_multiline_adjlist(G, "test.adjlist")
```

The path can be a file handle or a string with the name of the file. If a file handle is provided, it has to be opened in 'wb' mode.

```
>>> fh = open("test.adjlist", "wb")
>>> nx.write_multiline_adjlist(G, fh)
```

Filenames ending in .gz or .bz2 will be compressed.

```
>>> nx.write_multiline_adjlist(G, "test.adjlist.gz")
```

9.2.4 parse_multiline_adjlist

parse_multiline_adjlist(*lines*, *comments='#'*, *delimiter=None*, *create_using=None*, *nodetype=None*, *edgetype=None*)

Parse lines of a multiline adjacency list representation of a graph.

Parameters

lines
[list or iterator of strings] Input data in multiline adjlist format

create_using
[NetworkX graph constructor, optional (default=nx.Graph)] Graph type to create. If graph instance, then cleared before populated.

nodetype
[Python type, optional] Convert nodes to this type.

edgetype
[Python type, optional] Convert edges to this type.

comments
[string, optional] Marker for comment lines

delimiter
[string, optional] Separator for node labels. The default is whitespace.

Returns

G: NetworkX graph
The graph corresponding to the lines in multiline adjacency list format.

Examples

```
>>> lines = [
...     "1 2",
...     "2 {'weight':3, 'name': 'Frodo'}",
...     "3 {}",
...     "2 1",
...     "5 {'weight':6, 'name': 'Saruman'}",
... ]
>>> G = nx.parse_multiline_adjlist(iter(lines), nodetype=int)
>>> list(G)
[1, 2, 3, 5]
```

9.2.5 generate_multiline_adjlist

generate_multiline_adjlist(*G*, *delimiter=' '*)

Generate a single line of the graph G in multiline adjacency list format.

Parameters

G
[NetworkX graph]

delimiter
[string, optional] Separator for node labels

Returns

lines
[string] Lines of data in multiline adjlist format.

See also:

write_multiline_adjlist, ***read_multiline_adjlist***

Examples

```
>>> G = nx.lollipop_graph(4, 3)
>>> for line in nx.generate_multiline_adjlist(G):
...     print(line)
0 3
1 {}
2 {}
3 {}
1 2
2 {}
3 {}
2 1
3 {}
3 1
4 {}
4 1
5 {}
5 1
6 {}
6 0
```

9.3 Edge List

Read and write NetworkX graphs as edge lists.

The multi-line adjacency list format is useful for graphs with nodes that can be meaningfully represented as strings. With the edgelist format simple edge data can be stored but node or graph data is not. There is no way of representing isolated nodes unless the node has a self-loop edge.

9.3.1 Format

You can read or write three formats of edge lists with these functions.

Node pairs with no data:

```
1 2
```

Python dictionary as data:

```
1 2 {'weight':7, 'color':'green'}
```

Arbitrary data:

```
1 2 7 green
```

read_edgelist(path[, comments, delimiter, ...])	Read a graph from a list of edges.
write_edgelist(G, path[, comments, ...])	Write graph as a list of edges.
read_weighted_edgelist(path[, comments, ...])	Read a graph as list of edges with numeric weights.
write_weighted_edgelist(G, path[, comments, ...])	Write graph G as a list of edges with numeric weights.
generate_edgelist(G[, delimiter, data])	Generate a single line of the graph G in edge list format.
parse_edgelist(lines[, comments, delimiter, ...])	Parse lines of an edge list representation of a graph.

9.3.2 read_edgelist

read_edgelist(*path*, *comments='#'*, *delimiter=None*, *create_using=None*, *nodetype=None*, *data=True*, *edgetype=None*, *encoding='utf-8'*)

Read a graph from a list of edges.

Parameters

path
[file or string] File or filename to read. If a file is provided, it must be opened in 'rb' mode. Filenames ending in .gz or .bz2 will be uncompressed.

comments
[string, optional] The character used to indicate the start of a comment. To specify that no character should be treated as a comment, use `comments=None`.

delimiter
[string, optional] The string used to separate values. The default is whitespace.

create_using
[NetworkX graph constructor, optional (default=nx.Graph)] Graph type to create. If graph instance, then cleared before populated.

nodetype
[int, float, str, Python type, optional] Convert node data from strings to specified type

data
[bool or list of (label,type) tuples] Tuples specifying dictionary key names and types for edge data

edgetype
[int, float, str, Python type, optional OBSOLETE] Convert edge data from strings to specified type and use as 'weight'

encoding: string, optional
Specify which encoding to use when reading file.

Returns

G
[graph] A networkx Graph or other type specified with create_using

See also:

parse_edgelist
write_edgelist

Notes

Since nodes must be hashable, the function nodetype must return hashable types (e.g. int, float, str, frozenset - or tuples of those, etc.)

Examples

```
>>> nx.write_edgelist(nx.path_graph(4), "test.edgelist")
>>> G = nx.read_edgelist("test.edgelist")
```

```
>>> fh = open("test.edgelist", "rb")
>>> G = nx.read_edgelist(fh)
>>> fh.close()
```

```
>>> G = nx.read_edgelist("test.edgelist", nodetype=int)
>>> G = nx.read_edgelist("test.edgelist", create_using=nx.DiGraph)
```

Edgelist with data in a list:

```
>>> textline = "1 2 3"
>>> fh = open("test.edgelist", "w")
>>> d = fh.write(textline)
>>> fh.close()
>>> G = nx.read_edgelist("test.edgelist", nodetype=int, data=(("weight", float),))
>>> list(G)
[1, 2]
>>> list(G.edges(data=True))
[(1, 2, {'weight': 3.0})]
```

See parse_edgelist() for more examples of formatting.

9.3.3 write_edgelist

write_edgelist(*G*, *path*, *comments='#'*, *delimiter=' '*, *data=True*, *encoding='utf-8'*)

Write graph as a list of edges.

Parameters

G
[graph] A NetworkX graph

path
[file or string] File or filename to write. If a file is provided, it must be opened in 'wb' mode. Filenames ending in .gz or .bz2 will be compressed.

comments
[string, optional] The character used to indicate the start of a comment

delimiter
[string, optional] The string used to separate values. The default is whitespace.

data
[bool or list, optional] If False write no edge data. If True write a string representation of the edge data dictionary.. If a list (or other iterable) is provided, write the keys specified in the list.

encoding: string, optional
Specify which encoding to use when writing file.

See also:

read_edgelist
write_weighted_edgelist

Examples

```
>>> G = nx.path_graph(4)
>>> nx.write_edgelist(G, "test.edgelist")
>>> G = nx.path_graph(4)
>>> fh = open("test.edgelist", "wb")
>>> nx.write_edgelist(G, fh)
>>> nx.write_edgelist(G, "test.edgelist.gz")
>>> nx.write_edgelist(G, "test.edgelist.gz", data=False)
```

```
>>> G = nx.Graph()
>>> G.add_edge(1, 2, weight=7, color="red")
>>> nx.write_edgelist(G, "test.edgelist", data=False)
>>> nx.write_edgelist(G, "test.edgelist", data=["color"])
>>> nx.write_edgelist(G, "test.edgelist", data=["color", "weight"])
```

9.3.4 read_weighted_edgelist

read_weighted_edgelist(*path*, *comments='#'*, *delimiter=None*, *create_using=None*, *nodetype=None*, *encoding='utf-8'*)

Read a graph as list of edges with numeric weights.

Parameters

path
[file or string] File or filename to read. If a file is provided, it must be opened in 'rb' mode. Filenames ending in .gz or .bz2 will be uncompressed.

comments
[string, optional] The character used to indicate the start of a comment.

delimiter
[string, optional] The string used to separate values. The default is whitespace.

create_using
[NetworkX graph constructor, optional (default=nx.Graph)] Graph type to create. If graph instance, then cleared before populated.

nodetype
[int, float, str, Python type, optional] Convert node data from strings to specified type

encoding: string, optional
Specify which encoding to use when reading file.

Returns

G
[graph] A networkx Graph or other type specified with create_using

See also:

write_weighted_edgelist

Notes

Since nodes must be hashable, the function nodetype must return hashable types (e.g. int, float, str, frozenset - or tuples of those, etc.)

Example edgelist file format.

With numeric edge data:

```
# read with
# >>> G=nx.read_weighted_edgelist(fh)
# source target data
a b 1
a c 3.14159
d e 42
```

9.3.5 write_weighted_edgelist

write_weighted_edgelist(*G*, *path*, *comments='#'*, *delimiter=' '*, *encoding='utf-8'*)

Write graph G as a list of edges with numeric weights.

Parameters

G
[graph] A NetworkX graph

path
[file or string] File or filename to write. If a file is provided, it must be opened in 'wb' mode. Filenames ending in .gz or .bz2 will be compressed.

comments
[string, optional] The character used to indicate the start of a comment

delimiter
[string, optional] The string used to separate values. The default is whitespace.

encoding: string, optional
Specify which encoding to use when writing file.

See also:

read_edgelist
write_edgelist
read_weighted_edgelist

Examples

```
>>> G = nx.Graph()
>>> G.add_edge(1, 2, weight=7)
>>> nx.write_weighted_edgelist(G, "test.weighted.edgelist")
```

9.3.6 generate_edgelist

generate_edgelist(*G*, *delimiter=' '*, *data=True*)

Generate a single line of the graph G in edge list format.

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.

See also:

write_adjlist, read_adjlist

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
```

9.3.7 parse_edgelist

parse_edgelist(*lines*, *comments='#'*, *delimiter=None*, *create_using=None*, *nodetype=None*, *data=True*)

Parse lines of an edge list representation of a graph.

Parameters

lines
[list or iterator of strings] Input data in edgelist format

comments
[string, optional] Marker for comment lines. Default is `'#'`. To specify that no character

should be treated as a comment, use `comments=None`.

delimiter
[string, optional] Separator for node labels. Default is `None`, meaning any whitespace.

create_using
[NetworkX graph constructor, optional (default=nx.Graph)] Graph type to create. If graph instance, then cleared before populated.

nodetype
[Python type, optional] Convert nodes to this type. Default is `None`, meaning no conversion is performed.

data
[bool or list of (label,type) tuples] If `False` generate no edge data or if `True` use a dictionary representation of edge data or a list tuples specifying dictionary key names and types for edge data.

Returns

G: NetworkX Graph
The graph corresponding to lines

See also:

read_weighted_edgelist

Examples

Edgelist with no data:

```
>>> lines = ["1 2", "2 3", "3 4"]
>>> G = nx.parse_edgelist(lines, nodetype=int)
>>> list(G)
[1, 2, 3, 4]
>>> list(G.edges())
[(1, 2), (2, 3), (3, 4)]
```

Edgelist with data in Python dictionary representation:

```
>>> lines = ["1 2 {'weight': 3}", "2 3 {'weight': 27}", "3 4 {'weight': 3.0}"]
>>> G = nx.parse_edgelist(lines, nodetype=int)
>>> list(G)
[1, 2, 3, 4]
>>> list(G.edges(data=True))
[(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 = nx.parse_edgelist(lines, nodetype=int, data=(("weight", float),))
>>> list(G)
[1, 2, 3, 4]
>>> list(G.edges(data=True))
[(1, 2, {'weight': 3.0}), (2, 3, {'weight': 27.0}), (3, 4, {'weight': 3.0})]
```

9.4 GEXF

Read and write graphs in GEXF format.

Warning: This parser uses the standard xml library present in Python, which is insecure - see library/xml for additional information. Only parse GEFX files you trust.

GEXF (Graph Exchange XML Format) is a language for describing complex network structures, their associated data and dynamics.

This implementation does not support mixed graphs (directed and undirected edges together).

9.4.1 Format

GEXF is an XML format. See http://gexf.net/schema.html for the specification and http://gexf.net/basic.html for examples.

read_gexf(path[, node_type, relabel, version])	Read graph in GEXF format from path.
write_gexf(G, path[, encoding, prettyprint, ...])	Write G in GEXF format to path.
generate_gexf(G[, encoding, prettyprint, ...])	Generate lines of GEXF format representation of G.
relabel_gexf_graph(G)	Relabel graph using "label" node keyword for node label.

9.4.2 read_gexf

read_gexf(*path*, *node_type=None*, *relabel=False*, *version='1.2draft'*)

Read graph in GEXF format from path.

"GEXF (Graph Exchange XML Format) is a language for describing complex networks structures, their associated data and dynamics" [1].

Parameters

path
[file or string] File or file name to read. File names ending in .gz or .bz2 will be decompressed.

node_type: Python type (default: None)
Convert node ids to this type if not None.

relabel
[bool (default: False)] If True relabel the nodes to use the GEXF node "label" attribute instead of the node "id" attribute as the NetworkX node label.

version
[string (default: 1.2draft)]

Version of GEFX File Format (see http://gexf.net/schema.html)
Supported values: "1.1draft", "1.2draft"

Returns

graph: NetworkX graph
If no parallel edges are found a Graph or DiGraph is returned. Otherwise a MultiGraph or MultiDiGraph is returned.

Notes

This implementation does not support mixed graphs (directed and undirected edges together).

References

[1]

9.4.3 write_gexf

write_gexf(*G*, *path*, *encoding='utf-8'*, *prettyprint=True*, *version='1.2draft'*)

Write G in GEXF format to path.

"GEXF (Graph Exchange XML Format) is a language for describing complex networks structures, their associated data and dynamics" [1].

Node attributes are checked according to the version of the GEXF schemas used for parameters which are not user defined, e.g. visualization 'viz' [2]. See example for usage.

Parameters

G
[graph] A NetworkX graph

path
[file or string] File or file name to write. File names ending in .gz or .bz2 will be compressed.

encoding
[string (optional, default: 'utf-8')] Encoding for text data.

prettyprint
[bool (optional, default: True)] If True use line breaks and indenting in output XML.

version: string (optional, default: '1.2draft')
The version of GEXF to be used for nodes attributes checking

Notes

This implementation does not support mixed graphs (directed and undirected edges together).

The node id attribute is set to be the string of the node label. If you want to specify an id use set it as node data, e.g. node['a']['id']=1 to set the id of node 'a' to 1.

References

[1], [2]

Examples

```
>>> G = nx.path_graph(4)
>>> nx.write_gexf(G, "test.gexf")
```

visualization data >>> G.nodes[0]["viz"] = {"size": 54} >>> G.nodes[0]["viz"]["position"] = {"x": 0, "y": 1} >>> G.nodes[0]["viz"]["color"] = {"r": 0, "g": 0, "b": 256}

9.4.4 generate_gexf

generate_gexf(*G*, *encoding='utf-8'*, *prettyprint=True*, *version='1.2draft'*)

Generate lines of GEXF format representation of G.

"GEXF (Graph Exchange XML Format) is a language for describing complex networks structures, their associated data and dynamics" [1].

Parameters

G
[graph]

A NetworkX graph
encoding
[string (optional, default: 'utf-8')]

Encoding for text data.
prettyprint
[bool (optional, default: True)]

If True use line breaks and indenting in output XML.
version
[string (default: 1.2draft)]

Version of GEFX File Format (see http://gexf.net/schema.html)
Supported values: "1.1draft", "1.2draft"

Notes

This implementation does not support mixed graphs (directed and undirected edges together).

The node id attribute is set to be the string of the node label. If you want to specify an id use set it as node data, e.g. node['a']['id']=1 to set the id of node 'a' to 1.

References

[1]

Examples

```
>>> G = nx.path_graph(4)
>>> linefeed = chr(10)  # linefeed=
```

```
>>> s = linefeed.join(nx.generate_gexf(G))
>>> for line in nx.generate_gexf(G):
...     print(line)
```

9.4.5 relabel_gexf_graph

relabel_gexf_graph(*G*)

Relabel graph using "label" node keyword for node label.

Parameters

G
[graph] A NetworkX graph read from GEXF data

Returns

H
[graph] A NetworkX graph with relabeled nodes

Raises

NetworkXError
If node labels are missing or not unique while relabel=True.

Notes

This function relabels the nodes in a NetworkX graph with the "label" attribute. It also handles relabeling the specific GEXF node attributes "parents", and "pid".

9.5 GML

Read graphs in GML format.

"GML, the Graph Modelling Language, is our proposal for a portable file format for graphs. GML's key features are portability, simple syntax, extensibility and flexibility. A GML file consists of a hierarchical key-value lists. Graphs can be annotated with arbitrary data structures. The idea for a common file format was born at the GD'95; this proposal is the outcome of many discussions. GML is the standard file format in the Graphlet graph editor system. It has been overtaken and adapted by several other systems for drawing graphs."

GML files are stored using a 7-bit ASCII encoding with any extended ASCII characters (iso8859-1) appearing as HTML character entities. You will need to give some thought into how the exported data should interact with different languages and even different Python versions. Re-importing from gml is also a concern.

Without specifying a `stringizer`/`destringizer`, the code is capable of writing `int`/`float`/`str`/`dict`/`list` data as required by the GML specification. For writing other data types, and for reading data other than `str` you need to explicitly supply a `stringizer`/`destringizer`.

For additional documentation on the GML file format, please see the GML website.

Several example graphs in GML format may be found on Mark Newman's Network data page.

read_gml(path[, label, destringizer])	Read graph in GML format from `path`.
write_gml(G, path[, stringizer])	Write a graph `G` in GML format to the file or file handle `path`.
parse_gml(lines[, label, destringizer])	Parse GML graph from a string or iterable.
generate_gml(G[, stringizer])	Generate a single entry of the graph `G` in GML format.
literal_destringizer(rep)	Convert a Python literal to the value it represents.
literal_stringizer(value)	Convert a `value` to a Python literal in GML representation.

9.5.1 read_gml

read_gml(*path*, *label='label'*, *destringizer=None*)

Read graph in GML format from `path`.

Parameters

path
[filename or filehandle] The filename or filehandle to read from.

label
[string, optional] If not None, the parsed nodes will be renamed according to node attributes indicated by `label`. Default value: 'label'.

destringizer
[callable, optional] A `destringizer` that recovers values stored as strings in GML. If it cannot convert a string to a value, a `ValueError` is raised. Default value : None.

Returns

G
[NetworkX graph] The parsed graph.

Raises

NetworkXError
If the input cannot be parsed.

See also:

write_gml*, *parse_gml
literal_destringizer

Notes

GML files are stored using a 7-bit ASCII encoding with any extended ASCII characters (iso8859-1) appearing as HTML character entities. Without specifying a `stringizer`/`destringizer`, the code is capable of writing `int`/`float`/`str`/`dict`/`list` data as required by the GML specification. For writing other data types, and for reading data other than `str` you need to explicitly supply a `stringizer`/`destringizer`.

For additional documentation on the GML file format, please see the GML url.

See the module docstring *networkx.readwrite.gml* for more details.

Examples

```
>>> G = nx.path_graph(4)
>>> nx.write_gml(G, "test.gml")
```

GML values are interpreted as strings by default:

```
>>> H = nx.read_gml("test.gml")
>>> H.nodes
NodeView(('0', '1', '2', '3'))
```

When a `destringizer` is provided, GML values are converted to the provided type. For example, integer nodes can be recovered as shown below:

```
>>> J = nx.read_gml("test.gml", destringizer=int)
>>> J.nodes
NodeView((0, 1, 2, 3))
```

9.5.2 write_gml

write_gml(*G*, *path*, *stringizer=None*)

Write a graph `G` in GML format to the file or file handle `path`.

Parameters

G
[NetworkX graph] The graph to be converted to GML.

path
[filename or filehandle] The filename or filehandle to write. Files whose names end with .gz or .bz2 will be compressed.

stringizer
[callable, optional] A `stringizer` which converts non-int/non-float/non-dict values into strings. If it cannot convert a value into a string, it should raise a `ValueError` to indicate that. Default value: None.

Raises

NetworkXError
If `stringizer` cannot convert a value into a string, or the value to convert is not a string while `stringizer` is None.

See also:

read_gml*, *generate_gml
literal_stringizer

Notes

Graph attributes named 'directed', 'multigraph', 'node' or 'edge', node attributes named 'id' or 'label', edge attributes named 'source' or 'target' (or 'key' if `G` is a multigraph) are ignored because these attribute names are used to encode the graph structure.

GML files are stored using a 7-bit ASCII encoding with any extended ASCII characters (iso8859-1) appearing as HTML character entities. Without specifying a `stringizer`/`destringizer`, the code is capable of writing `int`/`float`/`str`/`dict`/`list` data as required by the GML specification. For writing other data types, and for reading data other than `str` you need to explicitly supply a `stringizer`/`destringizer`.

Note that while we allow non-standard GML to be read from a file, we make sure to write GML format. In particular, underscores are not allowed in attribute names. For additional documentation on the GML file format, please see the GML url.

See the module docstring *`networkx.readwrite.gml`* for more details.

Examples

```
>>> G = nx.path_graph(4)
>>> nx.write_gml(G, "test.gml")
```

Filenames ending in .gz or .bz2 will be compressed.

```
>>> nx.write_gml(G, "test.gml.gz")
```

9.5.3 parse_gml

parse_gml(*lines*, *label='label'*, *destringizer=None*)

Parse GML graph from a string or iterable.

Parameters

lines
[string or iterable of strings] Data in GML format.

label
[string, optional] If not None, the parsed nodes will be renamed according to node attributes indicated by `label`. Default value: 'label'.

destringizer
[callable, optional] A `destringizer` that recovers values stored as strings in GML. If it cannot convert a string to a value, a `ValueError` is raised. Default value : None.

Returns

G
[NetworkX graph] The parsed graph.

Raises

NetworkXError
If the input cannot be parsed.

See also:

`write_gml`*, *`read_gml`

Notes

This stores nested GML attributes as dictionaries in the NetworkX graph, node, and edge attribute structures.

GML files are stored using a 7-bit ASCII encoding with any extended ASCII characters (iso8859-1) appearing as HTML character entities. Without specifying a `stringizer`/`destringizer`, the code is capable of writing `int`/`float`/`str`/`dict`/`list` data as required by the GML specification. For writing other data types, and for reading data other than `str` you need to explicitly supply a `stringizer`/`destringizer`.

For additional documentation on the GML file format, please see the GML url.

See the module docstring *networkx.readwrite.gml* for more details.

9.5.4 generate_gml

generate_gml(*G*, *stringizer=None*)

Generate a single entry of the graph `G` in GML format.

Parameters

G
[NetworkX graph] The graph to be converted to GML.

stringizer
[callable, optional] A `stringizer` which converts non-int/non-float/non-dict values into strings. If it cannot convert a value into a string, it should raise a `ValueError` to indicate that. Default value: None.

Returns

lines: generator of strings
Lines of GML data. Newlines are not appended.

Raises

NetworkXError
If `stringizer` cannot convert a value into a string, or the value to convert is not a string while `stringizer` is None.

See also:

literal_stringizer

Notes

Graph attributes named 'directed', 'multigraph', 'node' or 'edge', node attributes named 'id' or 'label', edge attributes named 'source' or 'target' (or 'key' if `G` is a multigraph) are ignored because these attribute names are used to encode the graph structure.

GML files are stored using a 7-bit ASCII encoding with any extended ASCII characters (iso8859-1) appearing as HTML character entities. Without specifying a `stringizer`/`destringizer`, the code is capable of writing `int`/`float`/`str`/`dict`/`list` data as required by the GML specification. For writing other data types, and for reading data other than `str` you need to explicitly supply a `stringizer`/`destringizer`.

For additional documentation on the GML file format, please see the GML url.

See the module docstring *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
  ]
]
```

9.5.5 literal_destringizer

literal_destringizer(*rep*)

Convert a Python literal to the value it represents.

Parameters

rep
[string] A Python literal.

Returns

value
[object] The value of the Python literal.

Raises

ValueError
If `rep` is not a Python literal.

9.5.6 literal_stringizer

literal_stringizer(*value*)

Convert a `value` to a Python literal in GML representation.

Parameters

value
[object] The `value` to be converted to GML representation.

Returns

rep
[string] A double-quoted Python literal representing value. Unprintable characters are replaced by XML character references.

Raises

ValueError
If `value` cannot be converted to GML.

Notes

literal_stringizer is largely the same as `repr` in terms of functionality but attempts prefix `unicode` and `bytes` literals with `u` and `b` to provide better interoperability of data generated by Python 2 and Python 3.

The original value can be recovered using the *networkx.readwrite.gml.literal_destringizer()* function.

9.6 GraphML

Read and write graphs in GraphML format.

Warning: This parser uses the standard xml library present in Python, which is insecure - see library/xml for additional information. Only parse GraphML files you trust.

This implementation does not support mixed graphs (directed and unidirected edges together), hyperedges, nested graphs, or ports.

"GraphML is a comprehensive and easy-to-use file format for graphs. It consists of a language core to describe the structural properties of a graph and a flexible extension mechanism to add application-specific data. Its main features include support of

- directed, undirected, and mixed graphs,
- hypergraphs,
- hierarchical graphs,
- graphical representations,
- references to external data,
- application-specific attribute data, and
- light-weight parsers.

Unlike many other file formats for graphs, GraphML does not use a custom syntax. Instead, it is based on XML and hence ideally suited as a common denominator for all kinds of services generating, archiving, or processing graphs."

http://graphml.graphdrawing.org/

9.6.1 Format

GraphML is an XML format. See http://graphml.graphdrawing.org/specification.html for the specification and http://graphml.graphdrawing.org/primer/graphml-primer.html for examples.

read_graphml(path[, node_type, ...])	Read graph in GraphML format from path.
write_graphml(G, path[, encoding, ...])	Write G in GraphML XML format to path
generate_graphml(G[, encoding, prettyprint, ...])	Generate GraphML lines for G
parse_graphml(graphml_string[, node_type, ...])	Read graph in GraphML format from string.

9.6.2 read_graphml

read_graphml(*path*, *node_type=<class 'str'>*, *edge_key_type=<class 'int'>*, *force_multigraph=False*)

Read graph in GraphML format from path.

Parameters

path
[file or string] File or filename to write. Filenames ending in .gz or .bz2 will be compressed.

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 parallel edges are present or `force_multigraph=True` then a MultiGraph or MultiDiGraph is returned. Otherwise a Graph/DiGraph. The returned graph is directed if the file indicates it should be.

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"]
>>> for node, data in G.nodes(data=True):
...     if "color" not in data:
...         data["color"] = default_color
>>> default_color = G.graph["edge_default"]["color"]
>>> for u, v, data in G.edges(data=True):
```

(continues on next page)

(continued from previous page)

```
...     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".

9.6.3 write_graphml

write_graphml(*G*, *path*, *encoding='utf-8'*, *prettyprint=True*, *infer_numeric_types=False*, *named_key_ids=False*, *edge_id_from_attribute=None*)

Write G in GraphML XML format to path

This function uses the LXML framework and should be faster than the version using the xml library.

Parameters

G
[graph] A networkx graph

path
[file or string] File or filename to write. Filenames ending in .gz or .bz2 will be compressed.

encoding
[string (optional)] Encoding for text data.

prettyprint
[bool (optional)] If True use line breaks and indenting in output XML.

infer_numeric_types
[boolean] Determine if numeric types should be generalized. For example, if edges have both int and float 'weight' attributes, we infer in GraphML that both are floats.

named_key_ids
[bool (optional)] If True use attr.name as value for key elements' id attribute.

edge_id_from_attribute
[dict key (optional)] If provided, the graphml edge id is set by looking up the corresponding edge data attribute keyed by this parameter. If `None` or the key does not exist in edge data, the edge id is set by the edge key if `G` is a MultiGraph, else the edge id is left unset.

Notes

This implementation does not support mixed graphs (directed and unidirected edges together) hyperedges, nested graphs, or ports.

Examples

```
>>> G = nx.path_graph(4)
>>> nx.write_graphml_lxml(G, "fourpath.graphml")
```

9.6.4 generate_graphml

generate_graphml(*G*, *encoding='utf-8'*, *prettyprint=True*, *named_key_ids=False*, *edge_id_from_attribute=None*)

Generate GraphML lines for G

Parameters

G
[graph] A networkx graph

encoding
[string (optional)] Encoding for text data.

prettyprint
[bool (optional)] If True use line breaks and indenting in output XML.

named_key_ids
[bool (optional)] If True use attr.name as value for key elements' id attribute.

edge_id_from_attribute
[dict key (optional)] If provided, the graphml edge id is set by looking up the corresponding edge data attribute keyed by this parameter. If `None` or the key does not exist in edge data, the edge id is set by the edge key if `G` is a MultiGraph, else the edge id is left unset.

Notes

This implementation does not support mixed graphs (directed and unidirected edges together) hyperedges, nested graphs, or ports.

Examples

```
>>> G = nx.path_graph(4)
>>> linefeed = chr(10)  # linefeed =
```

```
>>> s = linefeed.join(nx.generate_graphml(G))
>>> for line in nx.generate_graphml(G):
...     print(line)
```

9.6.5 parse_graphml

parse_graphml(*graphml_string*, *node_type=<class 'str'>*, *edge_key_type=<class '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.

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"]
>>> for node, data in G.nodes(data=True):
...     if "color" not in data:
...         data["color"] = default_color
>>> default_color = G.graph["edge_default"]["color"]
>>> for u, v, data in G.edges(data=True):
...     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.

Examples

```
>>> G = nx.path_graph(4)
>>> linefeed = chr(10)  # linefeed =
```

```
>>> s = linefeed.join(nx.generate_graphml(G))
>>> H = nx.parse_graphml(s)
```

9.7 JSON

Generate and parse JSON serializable data for NetworkX graphs.

These formats are suitable for use with the d3.js examples https://d3js.org/

The three formats that you can generate with NetworkX are:

- node-link like in the d3.js example https://bl.ocks.org/mbostock/4062045
- tree like in the d3.js example https://bl.ocks.org/mbostock/4063550
- adjacency like in the d3.js example https://bost.ocks.org/mike/miserables/

node_link_data(G[, attrs, source, target, ...])	Returns data in node-link format that is suitable for JSON serialization and use in Javascript documents.
node_link_graph(data[, directed, ...])	Returns graph from node-link data format.
adjacency_data(G[, attrs])	Returns data in adjacency format that is suitable for JSON serialization and use in Javascript documents.
adjacency_graph(data[, directed, ...])	Returns graph from adjacency data format.
cytoscape_data(G[, name, ident])	Returns data in Cytoscape JSON format (cyjs).
cytoscape_graph(data[, name, ident])	Create a NetworkX graph from a dictionary in cytoscape JSON format.
tree_data(G, root[, ident, children])	Returns data in tree format that is suitable for JSON serialization and use in Javascript documents.
tree_graph(data[, ident, children])	Returns graph from tree data format.

9.7.1 node_link_data

node_link_data(*G*, *attrs=None*, ***, *source='source'*, *target='target'*, *name='id'*, *key='key'*, *link='links'*)

Returns data in node-link format that is suitable for JSON serialization and use in Javascript documents.

Parameters

G

[NetworkX graph]

attrs

[dict] A dictionary that contains five keys 'source', 'target', 'name', 'key' and 'link'. The corresponding values provide the attribute names for storing NetworkX-internal graph data. The values should be unique. Default value:

```
dict(source='source', target='target', name='id',
     key='key', link='links')
```

If some user-defined graph data use these attribute names as data keys, they may be silently dropped.

Deprecated since version 2.8.6: The `attrs` keyword argument will be replaced with `source`, `target`, `name`, `key` and `link`. in networkx 3.2

If the `attrs` keyword and the new keywords are both used in a single function call (not recommended) the `attrs` keyword argument will take precedence.

The values of the keywords must be unique.

source
[string] A string that provides the 'source' attribute name for storing NetworkX-internal graph data.

target
[string] A string that provides the 'target' attribute name for storing NetworkX-internal graph data.

name
[string] A string that provides the 'name' attribute name for storing NetworkX-internal graph data.

key
[string] A string that provides the 'key' attribute name for storing NetworkX-internal graph data.

link
[string] A string that provides the 'link' attribute name for storing NetworkX-internal graph data.

Returns

data
[dict] A dictionary with node-link formatted data.

Raises

NetworkXError
If the values of 'source', 'target' and 'key' are not unique.

See also:

node_link_graph, *adjacency_data*, *tree_data*

Notes

Graph, node, and link attributes are stored in this format. Note that attribute keys will be converted to strings in order to comply with JSON.

Attribute 'key' is only used for multigraphs.

To use *node_link_data* in conjunction with *node_link_graph*, the keyword names for the attributes must match.

Examples

```
>>> G = nx.Graph([("A", "B")])
>>> data1 = nx.node_link_data(G)
>>> data1
{'directed': False, 'multigraph': False, 'graph': {}, 'nodes': [{'id': 'A'}, {'id
↪': 'B'}], 'links': [{'source': 'A', 'target': 'B'}]}
```

To serialize with JSON

```
>>> import json
>>> s1 = json.dumps(data1)
>>> s1
```

(continues on next page)

(continued from previous page)

```
'{"directed": false, "multigraph": false, "graph": {}, "nodes": [{"id": "A"}, {"id
↪": "B"}], "links": [{"source": "A", "target": "B"}]}'
```

A graph can also be serialized by passing *node_link_data* as an encoder function. The two methods are equivalent.

```
>>> s1 = json.dumps(G, default=nx.node_link_data)
>>> s1
'{"directed": false, "multigraph": false, "graph": {}, "nodes": [{"id": "A"}, {"id
↪": "B"}], "links": [{"source": "A", "target": "B"}]}'
```

The attribute names for storing NetworkX-internal graph data can be specified as keyword options.

```
>>> H = nx.gn_graph(2)
>>> data2 = nx.node_link_data(H, link="edges", source="from", target="to")
>>> data2
{'directed': True, 'multigraph': False, 'graph': {}, 'nodes': [{'id': 0}, {'id':␣
↪1}], 'edges': [{'from': 1, 'to': 0}]}
```

9.7.2 node_link_graph

node_link_graph(*data*, *directed=False*, *multigraph=True*, *attrs=None*, ***, *source='source'*, *target='target'*, *name='id'*, *key='key'*, *link='links'*)

Returns graph from node-link data format. Useful for de-serialization from JSON.

Parameters

data
[dict] node-link formatted graph data

directed
[bool] If True, and direction not specified in data, return a directed graph.

multigraph
[bool] If True, and multigraph not specified in data, return a multigraph.

attrs
[dict] A dictionary that contains five keys 'source', 'target', 'name', 'key' and 'link'. The corresponding values provide the attribute names for storing NetworkX-internal graph data. Default value:

dict(source='source', target='target', name='id',
key='key', link='links')

Deprecated since version 2.8.6: The `attrs` keyword argument will be replaced with the individual keywords: `source`, `target`, `name`, `key` and `link`. in networkx 3.2.

If the `attrs` keyword and the new keywords are both used in a single function call (not recommended) the `attrs` keyword argument will take precedence.

The values of the keywords must be unique.

source
[string] A string that provides the 'source' attribute name for storing NetworkX-internal graph data.

target
: [string] A string that provides the 'target' attribute name for storing NetworkX-internal graph data.

name
: [string] A string that provides the 'name' attribute name for storing NetworkX-internal graph data.

key
: [string] A string that provides the 'key' attribute name for storing NetworkX-internal graph data.

link
: [string] A string that provides the 'link' attribute name for storing NetworkX-internal graph data.

Returns

G
: [NetworkX graph] A NetworkX graph object

See also:

node_link_data, *adjacency_data*, *tree_data*

Notes

Attribute 'key' is only used for multigraphs.

To use *node_link_data* in conjunction with *node_link_graph*, the keyword names for the attributes must match.

Examples

Create data in node-link format by converting a graph.

```
>>> G = nx.Graph([('A', 'B')])
>>> data = nx.node_link_data(G)
>>> data
{'directed': False, 'multigraph': False, 'graph': {}, 'nodes': [{'id': 'A'}, {'id
↪': 'B'}], 'links': [{'source': 'A', 'target': 'B'}]}
```

Revert data in node-link format to a graph.

```
>>> H = nx.node_link_graph(data)
>>> print(H.edges)
[('A', 'B')]
```

To serialize and deserialize a graph with JSON,

```
>>> import json
>>> d = json.dumps(node_link_data(G))
>>> H = node_link_graph(json.loads(d))
>>> print(G.edges, H.edges)
[('A', 'B')] [('A', 'B')]
```

9.7.3 adjacency_data

adjacency_data(*G*, *attrs={'id': 'id', 'key': 'key'}*)

Returns data in adjacency format that is suitable for JSON serialization and use in Javascript documents.

Parameters

G
[NetworkX graph]

attrs
[dict] A dictionary that contains two keys 'id' and 'key'. The corresponding values provide the attribute names for storing NetworkX-internal graph data. The values should be unique. Default value: `dict(id='id', key='key')`.

If some user-defined graph data use these attribute names as data keys, they may be silently dropped.

Returns

data
[dict] A dictionary with adjacency formatted data.

Raises

NetworkXError
If values in attrs are not unique.

See also:

adjacency_graph, *node_link_data*, *tree_data*

Notes

Graph, node, and link attributes will be written when using this format but attribute keys must be strings if you want to serialize the resulting data with JSON.

The default value of attrs will be changed in a future release of NetworkX.

Examples

```
>>> from networkx.readwrite import json_graph
>>> G = nx.Graph([(1, 2)])
>>> data = json_graph.adjacency_data(G)
```

To serialize with json

```
>>> import json
>>> s = json.dumps(data)
```

9.7.4 adjacency_graph

adjacency_graph(*data*, *directed=False*, *multigraph=True*, *attrs={'id': 'id', 'key': 'key'}*)

Returns graph from adjacency data format.

Parameters

data
[dict] Adjacency list formatted graph data

directed
[bool] If True, and direction not specified in data, return a directed graph.

multigraph
[bool] If True, and multigraph not specified in data, return a multigraph.

attrs
[dict] A dictionary that contains two keys 'id' and 'key'. The corresponding values provide the attribute names for storing NetworkX-internal graph data. The values should be unique. Default value: `dict(id='id', key='key')`.

Returns

G
[NetworkX graph] A NetworkX graph object

See also:

adjacency_graph, *node_link_data*, *tree_data*

Notes

The default value of attrs will be changed in a future release of NetworkX.

Examples

```
>>> from networkx.readwrite import json_graph
>>> G = nx.Graph([(1, 2)])
>>> data = json_graph.adjacency_data(G)
>>> H = json_graph.adjacency_graph(data)
```

9.7.5 cytoscape_data

cytoscape_data(*G*, *name='name'*, *ident='id'*)

Returns data in Cytoscape JSON format (cyjs).

Parameters

G
[NetworkX Graph] The graph to convert to cytoscape format

name
[string] A string which is mapped to the 'name' node element in cyjs format. Must not have the same value as `ident`.

ident
[string] A string which is mapped to the 'id' node element in cyjs format. Must not have the same value as `name`.

Returns

data: dict
A dictionary with cyjs formatted data.

Raises

NetworkXError
If the values for `name` and `ident` are identical.

See also:

`cytoscape_graph`
convert a dictionary in cyjs format to a graph

References

[1]

Examples

```
>>> G = nx.path_graph(2)
>>> nx.cytoscape_data(G)
{'data': [],
 'directed': False,
 'multigraph': False,
 'elements': {'nodes': [{'data': {'id': '0', 'value': 0, 'name': '0'}},
   {'data': {'id': '1', 'value': 1, 'name': '1'}}],
  'edges': [{'data': {'source': 0, 'target': 1}}]}}
```

9.7.6 cytoscape_graph

cytoscape_graph(*data*, *name='name'*, *ident='id'*)

Create a NetworkX graph from a dictionary in cytoscape JSON format.

Parameters

data
[dict] A dictionary of data conforming to cytoscape JSON format.

name
[string] A string which is mapped to the 'name' node element in cyjs format. Must not have the same value as `ident`.

ident
[string] A string which is mapped to the 'id' node element in cyjs format. Must not have the same value as `name`.

Returns

graph
[a NetworkX graph instance] The `graph` can be an instance of `Graph`, `DiGraph`, `MultiGraph`, or `MultiDiGraph` depending on the input data.

Raises

NetworkXError
If the `name` and `ident` attributes are identical.

See also:

cytoscape_data
convert a NetworkX graph to a dict in cyjs format

References

[1]

Examples

```
>>> data_dict = {
...     'data': [],
...     'directed': False,
...     'multigraph': False,
...     'elements': {'nodes': [{'data': {'id': '0', 'value': 0, 'name': '0'}},
...       {'data': {'id': '1', 'value': 1, 'name': '1'}}],
...      'edges': [{'data': {'source': 0, 'target': 1}}]}
... }
>>> G = nx.cytoscape_graph(data_dict)
>>> G.name
''
>>> G.nodes()
NodeView((0, 1))
>>> G.nodes(data=True)[0]
{'id': '0', 'value': 0, 'name': '0'}
>>> G.edges(data=True)
EdgeDataView([(0, 1, {'source': 0, 'target': 1})])
```

9.7.7 tree_data

tree_data(*G*, *root*, *ident='id'*, *children='children'*)

Returns data in tree format that is suitable for JSON serialization and use in Javascript documents.

Parameters

G
[NetworkX graph] G must be an oriented tree

root
[node] The root of the tree

ident
[string] Attribute name for storing NetworkX-internal graph data. `ident` must have a different value than `children`. The default is 'id'.

children
[string] Attribute name for storing NetworkX-internal graph data. `children` must have a different value than `ident`. The default is 'children'.

Returns

data
[dict] A dictionary with node-link formatted data.

Raises

NetworkXError
If `children` and `ident` attributes are identical.

See also:

tree_graph*, *node_link_data*, *adjacency_data

Notes

Node attributes are stored in this format but keys for attributes must be strings if you want to serialize with JSON.

Graph and edge attributes are not stored.

Examples

```
>>> from networkx.readwrite import json_graph
>>> G = nx.DiGraph([(1, 2)])
>>> data = json_graph.tree_data(G, root=1)
```

To serialize with json

```
>>> import json
>>> s = json.dumps(data)
```

9.7.8 tree_graph

tree_graph(*data*, *ident='id'*, *children='children'*)

Returns graph from tree data format.

Parameters

data
[dict] Tree formatted graph data

ident
[string] Attribute name for storing NetworkX-internal graph data. `ident` must have a different value than `children`. The default is 'id'.

children
[string] Attribute name for storing NetworkX-internal graph data. `children` must have a different value than `ident`. The default is 'children'.

Returns

G
[NetworkX DiGraph]

See also:

tree_data*, *node_link_data*, *adjacency_data

Examples

```
>>> from networkx.readwrite import json_graph
>>> G = nx.DiGraph([(1, 2)])
>>> data = json_graph.tree_data(G, root=1)
>>> H = json_graph.tree_graph(data)
```

9.8 LEDA

Read graphs in LEDA format.

LEDA is a C++ class library for efficient data types and algorithms.

9.8.1 Format

See http://www.algorithmic-solutions.info/leda_guide/graphs/leda_native_graph_fileformat.html

read_leda(path[, encoding])	Read graph in LEDA format from path.
parse_leda(lines)	Read graph in LEDA format from string or iterable.

9.8.2 read_leda

read_leda(*path*, *encoding='UTF-8'*)

Read graph in LEDA format from path.

Parameters

path
[file or string] File or filename to read. Filenames ending in .gz or .bz2 will be uncompressed.

Returns

G
[NetworkX graph]

References

[1]

Examples

G=nx.read_leda('file.leda')

9.8.3 parse_leda

parse_leda(*lines*)

Read graph in LEDA format from string or iterable.

Parameters

lines
[string or iterable] Data in LEDA format.

Returns

G
[NetworkX graph]

References

[1]

Examples

G=nx.parse_leda(string)

9.9 SparseGraph6

Functions for reading and writing graphs in the *graph6* or *sparse6* file formats.

According to the author of these formats,

> *graph6* and *sparse6* are formats for storing undirected graphs in a compact manner, using only printable ASCII characters. Files in these formats have text type and contain one line per graph.
>
> *graph6* is suitable for small graphs, or large dense graphs. *sparse6* is more space-efficient for large sparse graphs.
>
> ---graph6 and sparse6 homepage

9.9.1 Graph6

Functions for reading and writing graphs in the *graph6* format.

The *graph6* file format is suitable for small graphs or large dense graphs. For large sparse graphs, use the *sparse6* format.

For more information, see the graph6 homepage.

from_graph6_bytes(bytes_in)	Read a simple undirected graph in graph6 format from bytes.
read_graph6(path)	Read simple undirected graphs in graph6 format from path.
to_graph6_bytes(G[, nodes, header])	Convert a simple undirected graph to bytes in graph6 format.
write_graph6(G, path[, nodes, header])	Write a simple undirected graph to a path in graph6 format.

from_graph6_bytes

from_graph6_bytes(*bytes_in*)

Read a simple undirected graph in graph6 format from bytes.

Parameters

bytes_in
[bytes] Data in graph6 format, without a trailing newline.

Returns

G
[Graph]

Raises

NetworkXError
If bytes_in is unable to be parsed in graph6 format

ValueError
If any character `c` in bytes_in does not satisfy `63 <= ord(c) < 127`.

See also:

read_graph6, *write_graph6*

References

[1]

Examples

```
>>> G = nx.from_graph6_bytes(b"A_")
>>> sorted(G.edges())
[(0, 1)]
```

read_graph6

read_graph6(*path*)

Read simple undirected graphs in graph6 format from path.

Parameters

path
[file or string] File or filename to write.

Returns

G
[Graph or list of Graphs] If the file contains multiple lines then a list of graphs is returned

Raises

NetworkXError
If the string is unable to be parsed in graph6 format

See also:

from_graph6_bytes, *write_graph6*

References

[1]

Examples

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)]
```

to_graph6_bytes

to_graph6_bytes(*G*, *nodes=None*, *header=True*)

Convert a simple undirected graph to bytes in graph6 format.

Parameters

G
[Graph (undirected)]

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<<' bytes 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`.

See also:

from_graph6_bytes, *read_graph6*, **write_graph6_bytes**

Notes

The returned bytes end with a newline character.

The format does not support edge or node labels, parallel edges or self loops. If self loops are present they are silently ignored.

References

[1]

Examples

```
>>> nx.to_graph6_bytes(nx.path_graph(2))
b'>>graph6<<A_\n'
```

write_graph6

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`.

See also:

from_graph6_bytes, *read_graph6*

Notes

The function writes a newline character after writing the encoding of the graph.

The format does not support edge or node labels, parallel edges or self loops. If self loops are present they are silently ignored.

References

[1]

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'
```

9.9.2 Sparse6

Functions for reading and writing graphs in the *sparse6* format.

The *sparse6* file format is a space-efficient format for large sparse graphs. For small graphs or large dense graphs, use the *graph6* file format.

For more information, see the sparse6 homepage.

from_sparse6_bytes(string)	Read an undirected graph in sparse6 format from string.
read_sparse6(path)	Read an undirected graph in sparse6 format from path.
to_sparse6_bytes(G[, nodes, header])	Convert an undirected graph to bytes in sparse6 format.
write_sparse6(G, path[, nodes, header])	Write graph G to given path in sparse6 format.

from_sparse6_bytes

from_sparse6_bytes(*string*)

Read an undirected graph in sparse6 format from string.

Parameters

string
[string] Data in sparse6 format

Returns

G
[Graph]

Raises

NetworkXError
If the string is unable to be parsed in sparse6 format

See also:

read_sparse6, *write_sparse6*

References

[1]

Examples

```
>>> G = nx.from_sparse6_bytes(b":A_")
>>> sorted(G.edges())
[(0, 1), (0, 1), (0, 1)]
```

read_sparse6

read_sparse6(*path*)

Read an undirected graph in sparse6 format from path.

Parameters

path
[file or string] File or filename to write.

Returns

G
[Graph/Multigraph or list of Graphs/MultiGraphs] If the file contains multiple lines then a list of graphs is returned

Raises

NetworkXError
If the string is unable to be parsed in sparse6 format

See also:

read_sparse6, *from_sparse6_bytes*

References

[1]

Examples

You can read a sparse6 file by giving the path to the file:

```
>>> import tempfile
>>> with tempfile.NamedTemporaryFile(delete=False) as f:
...     _ = f.write(b">>sparse6<<:An\n")
...     _ = f.seek(0)
...     G = nx.read_sparse6(f.name)
```

(continues on next page)

(continued from previous page)

```
>>> list(G.edges())
[(0, 1)]
```

You can also read a sparse6 file by giving an open file-like object:

```
>>> import tempfile
>>> with tempfile.NamedTemporaryFile() as f:
...     _ = f.write(b">>sparse6<<:An\n")
...     _ = f.seek(0)
...     G = nx.read_sparse6(f)
>>> list(G.edges())
[(0, 1)]
```

to_sparse6_bytes

to_sparse6_bytes(*G*, *nodes=None*, *header=True*)

Convert an undirected graph to bytes in sparse6 format.

Parameters

G
[Graph (undirected)]

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 '>>sparse6<<' bytes to head of data.

Raises

NetworkXNotImplemented
If the graph is directed.

ValueError
If the graph has at least `2 ** 36` nodes; the sparse6 format is only defined for graphs of order less than `2 ** 36`.

See also:

to_sparse6_bytes, ***read_sparse6***, **write_sparse6_bytes**

Notes

The returned bytes end with a newline character.

The format does not support edge or node labels.

References

[1]

Examples

```
>>> nx.to_sparse6_bytes(nx.path_graph(2))
b'>>sparse6<<:An\n'
```

write_sparse6

write_sparse6(*G*, *path*, *nodes=None*, *header=True*)

Write graph G to given path in sparse6 format.

Parameters

G
[Graph (undirected)]

path
[file or string] File or filename to write

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 '>>sparse6<<' string to head of data

Raises

NetworkXError
If the graph is directed

See also:

read_sparse6, *from_sparse6_bytes*

Notes

The format does not support edge or node labels.

References

[1]

Examples

You can write a sparse6 file by giving the path to the file:

```
>>> import tempfile
>>> with tempfile.NamedTemporaryFile(delete=False) as f:
...     nx.write_sparse6(nx.path_graph(2), f.name)
...     print(f.read())
b'>>sparse6<<:An\n'
```

You can also write a sparse6 file by giving an open file-like object:

```
>>> with tempfile.NamedTemporaryFile() as f:
...     nx.write_sparse6(nx.path_graph(2), f)
...     _ = f.seek(0)
...     print(f.read())
b'>>sparse6<<:An\n'
```

9.10 Pajek

Read graphs in Pajek format.

This implementation handles directed and undirected graphs including those with self loops and parallel edges.

9.10.1 Format

See http://vlado.fmf.uni-lj.si/pub/networks/pajek/doc/draweps.htm for format information.

read_pajek(path[, encoding])	Read graph in Pajek format from path.
write_pajek(G, path[, encoding])	Write graph in Pajek format to path.
parse_pajek(lines)	Parse Pajek format graph from string or iterable.
generate_pajek(G)	Generate lines in Pajek graph format.

9.10.2 read_pajek

read_pajek(*path*, *encoding='UTF-8'*)

Read graph in Pajek format from path.

Parameters

path

[file or string] File or filename to write. Filenames ending in .gz or .bz2 will be uncompressed.

Returns

G

[NetworkX MultiGraph or MultiDiGraph.]

References

See http://vlado.fmf.uni-lj.si/pub/networks/pajek/doc/draweps.htm for format information.

Examples

```
>>> G = nx.path_graph(4)
>>> nx.write_pajek(G, "test.net")
>>> G = nx.read_pajek("test.net")
```

To create a Graph instead of a MultiGraph use

```
>>> G1 = nx.Graph(G)
```

9.10.3 write_pajek

write_pajek(*G*, *path*, *encoding='UTF-8'*)

Write graph in Pajek format to path.

Parameters

G
[graph] A Networkx graph

path
[file or string] File or filename to write. Filenames ending in .gz or .bz2 will be compressed.

Warning: Optional node attributes and edge attributes must be non-empty strings. Otherwise it will not be written into the file. You will need to convert those attributes to strings if you want to keep them.

References

See http://vlado.fmf.uni-lj.si/pub/networks/pajek/doc/draweps.htm for format information.

Examples

```
>>> G = nx.path_graph(4)
>>> nx.write_pajek(G, "test.net")
```

9.10.4 parse_pajek

parse_pajek(*lines*)

Parse Pajek format graph from string or iterable.

Parameters

lines
[string or iterable] Data in Pajek format.

Returns

G
[NetworkX graph]

See also:

read_pajek

9.10.5 generate_pajek

generate_pajek(*G*)

Generate lines in Pajek graph format.

Parameters

G
[graph] A Networkx graph

References

See http://vlado.fmf.uni-lj.si/pub/networks/pajek/doc/draweps.htm for format information.

9.11 Matrix Market

The Matrix Market exchange format is a text-based file format described by NIST. Matrix Market supports both a **coordinate format** for sparse matrices and an **array format** for dense matrices. The `scipy.io` module provides the `scipy.io.mmread` and `scipy.io.mmwrite` functions to read and write data in Matrix Market format, respectively. These functions work with either `numpy.ndarray` or `scipy.sparse.coo_matrix` objects depending on whether the data is in **array** or **coordinate** format. These functions can be combined with those of NetworkX's *convert_matrix* module to read and write Graphs in Matrix Market format.

9.11.1 Examples

Reading and writing graphs using Matrix Market's **array format** for dense matrices:

```
>>> import scipy as sp
>>> import scipy.io  # for mmread() and mmwrite()
>>> import io  # Use BytesIO as a stand-in for a Python file object
>>> fh = io.BytesIO()

>>> G = nx.complete_graph(5)
>>> a = nx.to_numpy_array(G)
>>> print(a)
[[0. 1. 1. 1. 1.]
 [1. 0. 1. 1. 1.]
 [1. 1. 0. 1. 1.]
 [1. 1. 1. 0. 1.]
 [1. 1. 1. 1. 0.]]

>>> # Write to file in Matrix Market array format
>>> sp.io.mmwrite(fh, a)
>>> print(fh.getvalue().decode('utf-8'))  # file contents
```

(continues on next page)

(continued from previous page)

```
%%MatrixMarket matrix array real symmetric
%
5 5
0.0000000000000000e+00
1.0000000000000000e+00
1.0000000000000000e+00
1.0000000000000000e+00
1.0000000000000000e+00
0.0000000000000000e+00
1.0000000000000000e+00
1.0000000000000000e+00
1.0000000000000000e+00
0.0000000000000000e+00
1.0000000000000000e+00
1.0000000000000000e+00
0.0000000000000000e+00
1.0000000000000000e+00
0.0000000000000000e+00

>>> # Read from file
>>> fh.seek(0)
>>> H = nx.from_numpy_array(sp.io.mmread(fh))
>>> H.edges() == G.edges()
True
```

Reading and writing graphs using Matrix Market's **coordinate format** for sparse matrices:

```
>>> import scipy as sp
>>> import scipy.io  # for mmread() and mmwrite()
>>> import io  # Use BytesIO as a stand-in for a Python file object
>>> fh = io.BytesIO()

>>> G = nx.path_graph(5)
>>> m = nx.to_scipy_sparse_array(G)
>>> print(m)
  (0, 1)        1
  (1, 0)        1
  (1, 2)        1
  (2, 1)        1
  (2, 3)        1
  (3, 2)        1
  (3, 4)        1
  (4, 3)        1

>>> sp.io.mmwrite(fh, m)
>>> print(fh.getvalue().decode('utf-8'))  # file contents
%%MatrixMarket matrix coordinate integer symmetric
%
5 5 4
2 1 1
3 2 1
4 3 1
5 4 1

>>> # Read from file
>>> fh.seek(0)
>>> H = nx.from_scipy_sparse_array(sp.io.mmread(fh))
```

(continues on next page)

(continued from previous page)

```
>>> H.edges() == G.edges()
True
```

CHAPTER

TEN

DRAWING

NetworkX provides basic functionality for visualizing graphs, but its main goal is to enable graph analysis rather than perform graph visualization. In the future, graph visualization functionality may be removed from NetworkX or only available as an add-on package.

Proper graph visualization is hard, and we highly recommend that people visualize their graphs with tools dedicated to that task. Notable examples of dedicated and fully-featured graph visualization tools are Cytoscape, Gephi, Graphviz and, for LaTeX typesetting, PGF/TikZ. To use these and other such tools, you should export your NetworkX graph into a format that can be read by those tools. For example, Cytoscape can read the GraphML format, and so, `networkx.write_graphml(G, path)` might be an appropriate choice.

More information on the features provided here are available at

- matplotlib: http://matplotlib.org/
- pygraphviz: http://pygraphviz.github.io/

10.1 Matplotlib

Draw networks with matplotlib.

10.1.1 Examples

```
>>> G = nx.complete_graph(5)
>>> nx.draw(G)
```

10.1.2 See Also

- matplotlib
- `matplotlib.pyplot.scatter()`
- `matplotlib.patches.FancyArrowPatch`

draw(G[, pos, ax])	Draw the graph G with Matplotlib.
draw_networkx(G[, pos, arrows, with_labels])	Draw the graph G using Matplotlib.
draw_networkx_nodes(G, pos[, nodelist, ...])	Draw the nodes of the graph G.
draw_networkx_edges(G, pos[, edgelist, ...])	Draw the edges of the graph G.
draw_networkx_labels(G, pos[, labels, ...])	Draw node labels on the graph G.
draw_networkx_edge_labels(G, pos[, ...])	Draw edge labels.
draw_circular(G, **kwargs)	Draw the graph G with a circular layout.
draw_kamada_kawai(G, **kwargs)	Draw the graph G with a Kamada-Kawai force-directed layout.
draw_planar(G, **kwargs)	Draw a planar networkx graph G with planar layout.
draw_random(G, **kwargs)	Draw the graph G with a random layout.
draw_spectral(G, **kwargs)	Draw the graph G with a spectral 2D layout.
draw_spring(G, **kwargs)	Draw the graph G with a spring layout.
draw_shell(G[, nlist])	Draw networkx graph G with shell layout.

10.1.3 draw

draw(*G*, *pos=None*, *ax=None*, ***kwds*)

Draw the graph G with Matplotlib.

Draw the graph as a simple representation with no node labels or edge labels and using the full Matplotlib figure area and no axis labels by default. See draw_networkx() for more full-featured drawing that allows title, axis labels etc.

Parameters

G
[graph] A networkx graph

pos
[dictionary, optional] A dictionary with nodes as keys and positions as values. If not specified a spring layout positioning will be computed. See *networkx.drawing.layout* for functions that compute node positions.

ax
[Matplotlib Axes object, optional] Draw the graph in specified Matplotlib axes.

kwds
[optional keywords] See networkx.draw_networkx() for a description of optional keywords.

See also:

draw_networkx
draw_networkx_nodes
draw_networkx_edges
draw_networkx_labels
draw_networkx_edge_labels

Notes

This function has the same name as pylab.draw and pyplot.draw so beware when using `from networkx import *`

since you might overwrite the pylab.draw function.

With pyplot use

```
>>> import matplotlib.pyplot as plt
>>> G = nx.dodecahedral_graph()
>>> nx.draw(G)  # networkx draw()
>>> plt.draw()  # pyplot draw()
```

Also see the NetworkX drawing examples at https://networkx.org/documentation/latest/auto_examples/index.html

Examples

```
>>> G = nx.dodecahedral_graph()
>>> nx.draw(G)
>>> nx.draw(G, pos=nx.spring_layout(G))  # use spring layout
```

10.1.4 draw_networkx

draw_networkx(*G*, *pos=None*, *arrows=None*, *with_labels=True*, ***kwds*)

Draw the graph G using Matplotlib.

Draw the graph with Matplotlib with options for node positions, labeling, titles, and many other drawing features. See draw() for simple drawing without labels or axes.

Parameters

G
[graph] A networkx graph

pos
[dictionary, optional] A dictionary with nodes as keys and positions as values. If not specified a spring layout positioning will be computed. See *networkx.drawing.layout* for functions that compute node positions.

arrows
[bool or None, optional (default=None)] If `None`, directed graphs draw arrowheads with `FancyArrowPatch`, while undirected graphs draw edges via `LineCollection` for speed. If `True`, draw arrowheads with FancyArrowPatches (bendable and stylish). If `False`, draw edges using LineCollection (linear and fast). For directed graphs, if True draw arrowheads. Note: Arrows will be the same color as edges.

arrowstyle
[str (default='-|>' for directed graphs)] For directed graphs, choose the style of the arrowsheads. For undirected graphs default to '-'

See `matplotlib.patches.ArrowStyle` for more options.

arrowsize
[int or list (default=10)] For directed graphs, choose the size of the arrow head's length and width. A list of values can be passed in to assign a different size for arrow head's

length and width. See matplotlib.patches.FancyArrowPatch for attribute mutation_scale for more info.

with_labels
[bool (default=True)] Set to True to draw labels on the nodes.

ax
[Matplotlib Axes object, optional] Draw the graph in the specified Matplotlib axes.

nodelist
[list (default=list(G))] Draw only specified nodes

edgelist
[list (default=list(G.edges()))] Draw only specified edges

node_size
[scalar or array (default=300)] Size of nodes. If an array is specified it must be the same length as nodelist.

node_color
[color or array of colors (default='#1f78b4')] Node color. Can be a single color or a sequence of colors with the same length as nodelist. 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 cmap and vmin,vmax parameters. See matplotlib.scatter for more details.

node_shape
[string (default='o')] The shape of the node. Specification is as matplotlib.scatter 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
[string (default='k' black)] Font color string

font_weight
[string (default='normal')] Font weight

font_family
[string (default='sans-serif')] Font family

label
[string, optional] Label for graph legend

kwds
[optional keywords] See networkx.draw_networkx_nodes(), networkx.draw_networkx_edges(), and networkx.draw_networkx_labels() for a description of optional keywords.

See also:

draw
draw_networkx_nodes
draw_networkx_edges
draw_networkx_labels
draw_networkx_edge_labels

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

10.1.5 draw_networkx_nodes

draw_networkx_nodes(*G*, *pos*, *nodelist=None*, *node_size=300*, *node_color='#1f78b4'*, *node_shape='o'*, *alpha=None*, *cmap=None*, *vmin=None*, *vmax=None*, *ax=None*, *linewidths=None*, *edgecolors=None*, *label=None*, *margins=None*)

Draw the nodes of the graph G.

This draws only the nodes 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.

ax
[Matplotlib Axes object, optional] Draw the graph in the specified Matplotlib axes.

nodelist
[list (default list(G))] Draw only specified nodes

node_size
[scalar or array (default=300)] Size of nodes. If an array it must be the same length as nodelist.

node_color
[color or array of colors (default='#1f78b4')] Node color. Can be a single color or a sequence of colors with the same length as nodelist. 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 cmap and vmin,vmax parameters. See matplotlib.scatter for more details.

node_shape
[string (default='o')] The shape of the node. Specification is as matplotlib.scatter marker, one of 'so^>v<dph8'.

alpha
[float or array of floats (default=None)] The node transparency. This can be a single alpha value, in which case it will be applied to all the nodes of color. 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).

cmap
[Matplotlib colormap (default=None)] Colormap for mapping intensities of nodes

vmin,vmax
[floats or None (default=None)] Minimum and maximum for node colormap scaling

linewidths
[[None | scalar | sequence] (default=1.0)] Line width of symbol border

edgecolors
[[None | scalar | sequence] (default = node_color)] Colors of node borders

label
[[None | string]] Label for legend

margins
[float or 2-tuple, optional] Sets the padding for axis autoscaling. Increase margin to prevent clipping for nodes that are near the edges of an image. Values should be in the range `[0, 1]`. See `matplotlib.axes.Axes.margins()` for details. The default is `None`, which uses the Matplotlib default.

Returns

matplotlib.collections.PathCollection
`PathCollection` of the nodes.

See also:

draw
draw_networkx
draw_networkx_edges
draw_networkx_labels
draw_networkx_edge_labels

Examples

```
>>> G = nx.dodecahedral_graph()
>>> nodes = nx.draw_networkx_nodes(G, pos=nx.spring_layout(G))
```

Also see the NetworkX drawing examples at https://networkx.org/documentation/latest/auto_examples/index.html

10.1.6 draw_networkx_edges

draw_networkx_edges(*G*, *pos*, *edgelist=None*, *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*)

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 `FancyArrowPatch`, while undirected graphs draw edges via `LineCollection` for speed. If `True`, draw arrowheads with FancyArrowPatches (bendable and stylish). If `False`, draw edges using LineCollection (linear and fast).

Note: Arrowheads will be the same color as edges.

arrowstyle
[str (default='-|>' for directed graphs)] For directed graphs and `arrows==True` defaults to '-|>', For undirected graphs default to '-'.

See `matplotlib.patches.ArrowStyle` for more options.

arrowsize
[int (default=10)] For directed graphs, choose the size of the arrow head's length and width. See `matplotlib.patches.FancyArrowPatch` for attribute `mutation_scale` for more info.

connectionstyle
[string (default="arc3")] Pass the connectionstyle parameter to create curved arc of rounding radius rad. For example, connectionstyle='arc3,rad=0.2'. See `matplotlib.patches.ConnectionStyle` and `matplotlib.patches.FancyArrowPatch` for more info.

node_size
[scalar or array (default=300)] Size of nodes. Though the nodes are not drawn with this function, the node size is used in determining edge positioning.

nodelist
[list, optional (default=G.nodes())] This provides the node order for the `node_size` array (if it is an array).

node_shape
[string (default='o')] The marker used for nodes, used in determining edge positioning. Specification is as a `matplotlib.markers` marker, e.g. one of 'so^>v<dph8'.

label
[None or string] Label for legend

min_source_margin
[int (default=0)] The minimum margin (gap) at the begining of the edge at the source.

min_target_margin
[int (default=0)] The minimum margin (gap) at the end of the edge at the target.

Returns

matplotlib.colections.LineCollection or a list of matplotlib.patches.FancyArrowPatch
If `arrows=True`, a list of FancyArrowPatches is returned. If `arrows=False`, a LineCol-

lection is returned. If `arrows=None` (the default), then a LineCollection is returned if `G` is undirected, otherwise returns a list of FancyArrowPatches.

See also:

draw
draw_networkx
draw_networkx_nodes
draw_networkx_labels
draw_networkx_edge_labels

Notes

For directed graphs, arrows are drawn at the head end. Arrows can be turned off with keyword arrows=False or by passing an arrowstyle without an arrow on the end.

Be sure to include `node_size` as a keyword argument; arrows are drawn considering the size of nodes.

Self-loops are always drawn with `FancyArrowPatch` regardless of the value of `arrows` or whether `G` is directed. When `arrows=False` or `arrows=None` and `G` is undirected, the FancyArrowPatches corresponding to the self-loops are not explicitly returned. They should instead be accessed via the `Axes.patches` attribute (see examples).

Examples

```
>>> G = nx.dodecahedral_graph()
>>> edges = nx.draw_networkx_edges(G, pos=nx.spring_layout(G))
```

```
>>> G = nx.DiGraph()
>>> G.add_edges_from([(1, 2), (1, 3), (2, 3)])
>>> arcs = nx.draw_networkx_edges(G, pos=nx.spring_layout(G))
>>> alphas = [0.3, 0.4, 0.5]
>>> for i, arc in enumerate(arcs):  # change alpha values of arcs
...     arc.set_alpha(alphas[i])
```

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

10.1.7 draw_networkx_labels

draw_networkx_labels(*G*, *pos*, *labels=None*, *font_size=12*, *font_color='k'*, *font_family='sans-serif'*, *font_weight='normal'*, *alpha=None*, *bbox=None*, *horizontalalignment='center'*, *verticalalignment='center'*, *ax=None*, *clip_on=True*)

Draw node labels on 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.

labels
[dictionary (default={n: n for n in G})] Node labels in a dictionary of text labels keyed by node. Node-keys in labels should appear as keys in `pos`. If needed use: `{n:lab for n,lab in labels.items() if n in pos}`

font_size
[int (default=12)] Font size for text labels

font_color
[string (default='k' black)] Font color string

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

Returns

dict
`dict` of labels keyed on the nodes

See also:

draw
draw_networkx
draw_networkx_nodes
draw_networkx_edges
draw_networkx_edge_labels

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

10.1.8 draw_networkx_edge_labels

draw_networkx_edge_labels(*G*, *pos*, *edge_labels=None*, *label_pos=0.5*, *font_size=10*, *font_color='k'*, *font_family='sans-serif'*, *font_weight='normal'*, *alpha=None*, *bbox=None*, *horizontalalignment='center'*, *verticalalignment='center'*, *ax=None*, *rotate=True*, *clip_on=True*)

Draw edge labels.

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.

edge_labels
[dictionary (default=None)] Edge labels in a dictionary of labels keyed by edge two-tuple. Only labels for the keys in the dictionary are drawn.

label_pos
[float (default=0.5)] Position of edge label along edge (0=head, 0.5=center, 1=tail)

font_size
[int (default=10)] Font size for text labels

font_color
[string (default='k' black)] Font color string

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, optional] Specify text box properties (e.g. shape, color etc.) for edge labels. Default is {boxstyle='round', ec=(1.0, 1.0, 1.0), fc=(1.0, 1.0, 1.0)}.

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.

rotate
[bool (deafult=True)] Rotate edge labels to lie parallel to edges

clip_on
[bool (default=True)] Turn on clipping of edge labels at axis boundaries

Returns

dict
`dict` of labels keyed by edge

See also:

draw
draw_networkx
draw_networkx_nodes
draw_networkx_edges
draw_networkx_labels

Examples

```
>>> G = nx.dodecahedral_graph()
>>> edge_labels = nx.draw_networkx_edge_labels(G, pos=nx.spring_layout(G))
```

Also see the NetworkX drawing examples at https://networkx.org/documentation/latest/auto_examples/index.html

10.1.9 draw_circular

draw_circular(*G*, ***kwargs*)

Draw the graph `G` with a circular layout.

This is a convenience function equivalent to:

```
nx.draw(G, pos=nx.circular_layout(G), **kwargs)
```

Parameters

G
[graph] A networkx graph

kwargs
[optional keywords] See *draw_networkx* for a description of optional keywords.

See also:

circular_layout()

Notes

The layout is computed each time this function is called. For repeated drawing it is much more efficient to call *circular_layout* directly and reuse the result:

```
>>> G = nx.complete_graph(5)
>>> pos = nx.circular_layout(G)
>>> nx.draw(G, pos=pos)  # Draw the original graph
>>> # Draw a subgraph, reusing the same node positions
>>> nx.draw(G.subgraph([0, 1, 2]), pos=pos, node_color="red")
```

10.1.10 draw_kamada_kawai

draw_kamada_kawai(*G*, ***kwargs*)

Draw the graph G with a Kamada-Kawai force-directed layout.

This is a convenience function equivalent to:

```
nx.draw(G, pos=nx.kamada_kawai_layout(G), **kwargs)
```

Parameters

G
[graph] A networkx graph

kwargs
[optional keywords] See *draw_networkx* for a description of optional keywords.

See also:

kamada_kawai_layout()

Notes

The layout is computed each time this function is called. For repeated drawing it is much more efficient to call *kamada_kawai_layout* directly and reuse the result:

```
>>> G = nx.complete_graph(5)
>>> pos = nx.kamada_kawai_layout(G)
>>> nx.draw(G, pos=pos)  # Draw the original graph
>>> # Draw a subgraph, reusing the same node positions
>>> nx.draw(G.subgraph([0, 1, 2]), pos=pos, node_color="red")
```

10.1.11 draw_planar

draw_planar(*G*, ***kwargs*)

Draw a planar networkx graph G with planar layout.

This is a convenience function equivalent to:

```
nx.draw(G, pos=nx.planar_layout(G), **kwargs)
```

Parameters

G
[graph] A planar networkx graph

kwargs
[optional keywords] See *draw_networkx* for a description of optional keywords.

Raises

NetworkXException
When G is not planar

See also:

planar_layout()

Notes

The layout is computed each time this function is called. For repeated drawing it is much more efficient to call *planar_layout* directly and reuse the result:

```
>>> G = nx.path_graph(5)
>>> pos = nx.planar_layout(G)
>>> nx.draw(G, pos=pos)  # Draw the original graph
>>> # Draw a subgraph, reusing the same node positions
>>> nx.draw(G.subgraph([0, 1, 2]), pos=pos, node_color="red")
```

10.1.12 draw_random

draw_random(*G*, ***kwargs*)

Draw the graph G with a random layout.

This is a convenience function equivalent to:

```
nx.draw(G, pos=nx.random_layout(G), **kwargs)
```

Parameters

G
[graph] A networkx graph

kwargs
[optional keywords] See *draw_networkx* for a description of optional keywords.

See also:

random_layout()

Notes

The layout is computed each time this function is called. For repeated drawing it is much more efficient to call *random_layout* directly and reuse the result:

```
>>> G = nx.complete_graph(5)
>>> pos = nx.random_layout(G)
>>> nx.draw(G, pos=pos)  # Draw the original graph
>>> # Draw a subgraph, reusing the same node positions
>>> nx.draw(G.subgraph([0, 1, 2]), pos=pos, node_color="red")
```

10.1.13 draw_spectral

draw_spectral(*G*, ***kwargs*)

Draw the graph G with a spectral 2D layout.

This is a convenience function equivalent to:

```
nx.draw(G, pos=nx.spectral_layout(G), **kwargs)
```

For more information about how node positions are determined, see *spectral_layout*.

Parameters

G
[graph] A networkx graph

kwargs
[optional keywords] See *draw_networkx* for a description of optional keywords.

See also:

spectral_layout()

Notes

The layout is computed each time this function is called. For repeated drawing it is much more efficient to call *spectral_layout* directly and reuse the result:

```
>>> G = nx.complete_graph(5)
>>> pos = nx.spectral_layout(G)
>>> nx.draw(G, pos=pos)  # Draw the original graph
>>> # Draw a subgraph, reusing the same node positions
>>> nx.draw(G.subgraph([0, 1, 2]), pos=pos, node_color="red")
```

10.1.14 draw_spring

draw_spring(*G*, ***kwargs*)

Draw the graph `G` with a spring layout.

This is a convenience function equivalent to:

```
nx.draw(G, pos=nx.spring_layout(G), **kwargs)
```

Parameters

G
[graph] A networkx graph

kwargs
[optional keywords] See *draw_networkx* for a description of optional keywords.

See also:

draw
spring_layout()

Notes

spring_layout is also the default layout for *draw*, so this function is equivalent to *draw*.

The layout is computed each time this function is called. For repeated drawing it is much more efficient to call *spring_layout* directly and reuse the result:

```
>>> G = nx.complete_graph(5)
>>> pos = nx.spring_layout(G)
>>> nx.draw(G, pos=pos)  # Draw the original graph
>>> # Draw a subgraph, reusing the same node positions
>>> nx.draw(G.subgraph([0, 1, 2]), pos=pos, node_color="red")
```

10.1.15 draw_shell

draw_shell(*G*, *nlist=None*, ***kwargs*)

Draw networkx graph `G` with shell layout.

This is a convenience function equivalent to:

```
nx.draw(G, pos=nx.shell_layout(G, nlist=nlist), **kwargs)
```

Parameters

G
[graph] A networkx graph

nlist
[list of list of nodes, optional] A list containing lists of nodes representing the shells. Default is `None`, meaning all nodes are in a single shell. See *shell_layout* for details.

kwargs
[optional keywords] See *draw_networkx* for a description of optional keywords.

See also:

shell_layout()

Notes

The layout is computed each time this function is called. For repeated drawing it is much more efficient to call *shell_layout* directly and reuse the result:

```
>>> G = nx.complete_graph(5)
>>> pos = nx.shell_layout(G)
>>> nx.draw(G, pos=pos)  # Draw the original graph
>>> # Draw a subgraph, reusing the same node positions
>>> nx.draw(G.subgraph([0, 1, 2]), pos=pos, node_color="red")
```

10.2 Graphviz AGraph (dot)

Interface to pygraphviz AGraph class.

10.2.1 Examples

```
>>> G = nx.complete_graph(5)
>>> A = nx.nx_agraph.to_agraph(G)
>>> H = nx.nx_agraph.from_agraph(A)
```

10.2.2 See Also

- Pygraphviz: http://pygraphviz.github.io/
- Graphviz: https://www.graphviz.org
- DOT Language: http://www.graphviz.org/doc/info/lang.html

from_agraph(A[, create_using])	Returns a NetworkX Graph or DiGraph from a PyGraphviz graph.
to_agraph(N)	Returns a pygraphviz graph from a NetworkX graph N.
write_dot(G, path)	Write NetworkX graph G to Graphviz dot format on path.
read_dot(path)	Returns a NetworkX graph from a dot file on path.
graphviz_layout(G[, prog, root, args])	Create node positions for G using Graphviz.
pygraphviz_layout(G[, prog, root, args])	Create node positions for G using Graphviz.

10.2.3 from_agraph

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`.

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.

Examples

```
>>> K5 = nx.complete_graph(5)
>>> A = nx.nx_agraph.to_agraph(K5)
>>> G = nx.nx_agraph.from_agraph(A)
```

10.2.4 to_agraph

to_agraph(*N*)

Returns a pygraphviz graph from a NetworkX graph N.

Parameters

N
[NetworkX graph] A graph created with NetworkX

Notes

If N has an dict N.graph_attr an attempt will be made first to copy properties attached to the graph (see from_agraph) and then updated with the calling arguments if any.

Examples

```
>>> K5 = nx.complete_graph(5)
>>> A = nx.nx_agraph.to_agraph(K5)
```

10.2.5 write_dot

write_dot(*G*, *path*)

Write NetworkX graph G to Graphviz dot format on path.

Parameters

G
[graph] A networkx graph

path
[filename] Filename or file handle to write

Notes

To use a specific graph layout, call `A.layout` prior to *write_dot*. Note that some graphviz layouts are not guaranteed to be deterministic, see https://gitlab.com/graphviz/graphviz/-/issues/1767 for more info.

10.2.6 read_dot

read_dot(*path*)

Returns a NetworkX graph from a dot file on path.

Parameters

path
[file or string] File name or file handle to read.

10.2.7 graphviz_layout

graphviz_layout(*G*, *prog='neato'*, *root=None*, *args=''*)

Create node positions for G using Graphviz.

Parameters

G
[NetworkX graph] A graph created with NetworkX

prog
[string] Name of Graphviz layout program

root
[string, optional] Root node for twopi layout

args
[string, optional] Extra arguments to Graphviz layout program

Returns

Dictionary of x, y, positions keyed by node.

Notes

This is a wrapper for pygraphviz_layout.

Note that some graphviz layouts are not guaranteed to be deterministic, see https://gitlab.com/graphviz/graphviz/-/issues/1767 for more info.

Examples

```
>>> G = nx.petersen_graph()
>>> pos = nx.nx_agraph.graphviz_layout(G)
>>> pos = nx.nx_agraph.graphviz_layout(G, prog="dot")
```

10.2.8 pygraphviz_layout

pygraphviz_layout(*G*, *prog='neato'*, *root=None*, *args=''*)

Create node positions for G using Graphviz.

Parameters

G
[NetworkX graph] A graph created with NetworkX

prog
[string] Name of Graphviz layout program

root
[string, optional] Root node for twopi layout

args
[string, optional] Extra arguments to Graphviz layout program

Returns

node_pos
[dict] Dictionary of x, y, positions keyed by node.

Notes

If you use complex node objects, they may have the same string representation and GraphViz could treat them as the same node. The layout may assign both nodes a single location. See Issue #1568 If this occurs in your case, consider relabeling the nodes just for the layout computation using something similar to:

```
>>> H = nx.convert_node_labels_to_integers(G, label_attribute="node_label")
>>> H_layout = nx.nx_agraph.pygraphviz_layout(G, prog="dot")
>>> G_layout = {H.nodes[n]["node_label"]: p for n, p in H_layout.items()}
```

Note that some graphviz layouts are not guaranteed to be deterministic, see https://gitlab.com/graphviz/graphviz/-/issues/1767 for more info.

Examples

```
>>> G = nx.petersen_graph()
>>> pos = nx.nx_agraph.graphviz_layout(G)
>>> pos = nx.nx_agraph.graphviz_layout(G, prog="dot")
```

10.3 Graphviz with pydot

Import and export NetworkX graphs in Graphviz dot format using pydot.

Either this module or nx_agraph can be used to interface with graphviz.

10.3.1 Examples

```
>>> G = nx.complete_graph(5)
>>> PG = nx.nx_pydot.to_pydot(G)
>>> H = nx.nx_pydot.from_pydot(PG)
```

10.3.2 See Also

- pydot: https://github.com/erocarrera/pydot
- Graphviz: https://www.graphviz.org
- DOT Language: http://www.graphviz.org/doc/info/lang.html

from_pydot(P)	Returns a NetworkX graph from a Pydot graph.
to_pydot(N)	Returns a pydot graph from a NetworkX graph N.
write_dot(G, path)	Write NetworkX graph G to Graphviz dot format on path.
read_dot(path)	Returns a NetworkX `MultiGraph` or `MultiDiGraph` from the dot file with the passed path.
graphviz_layout(G[, prog, root])	Create node positions using Pydot and Graphviz.
pydot_layout(G[, prog, root])	Create node positions using `pydot` and Graphviz.

10.3.3 from_pydot

from_pydot(*P*)

Returns a NetworkX graph from a Pydot graph.

Parameters

P

[Pydot graph] A graph created with Pydot

Returns

G

[NetworkX multigraph] A MultiGraph or MultiDiGraph.

Examples

```
>>> K5 = nx.complete_graph(5)
>>> A = nx.nx_pydot.to_pydot(K5)
>>> G = nx.nx_pydot.from_pydot(A)   # return MultiGraph
```

make a Graph instead of MultiGraph >>> G = nx.Graph(nx.nx_pydot.from_pydot(A))

10.3.4 to_pydot

to_pydot(*N*)

Returns a pydot graph from a NetworkX graph N.

Parameters

N
[NetworkX graph] A graph created with NetworkX

Examples

```
>>> K5 = nx.complete_graph(5)
>>> P = nx.nx_pydot.to_pydot(K5)
```

10.3.5 write_dot

write_dot(*G*, *path*)

Write NetworkX graph G to Graphviz dot format on path.

Path can be a string or a file handle.

10.3.6 read_dot

read_dot(*path*)

Returns a NetworkX `MultiGraph` or `MultiDiGraph` from the dot file with the passed path.

If this file contains multiple graphs, only the first such graph is returned. All graphs _except_ the first are silently ignored.

Parameters

path
[str or file] Filename or file handle.

Returns

G
[MultiGraph or MultiDiGraph] A `MultiGraph` or `MultiDiGraph`.

Notes

Use `G = nx.Graph(nx.nx_pydot.read_dot(path))` to return a `Graph` instead of a `MultiGraph`.

10.3.7 graphviz_layout

graphviz_layout(*G*, *prog='neato'*, *root=None*)

Create node positions using Pydot and Graphviz.

Returns a dictionary of positions keyed by node.

Parameters

G
[NetworkX Graph] The graph for which the layout is computed.

prog
[string (default: 'neato')] The name of the GraphViz program to use for layout. Options depend on GraphViz version but may include: 'dot', 'twopi', 'fdp', 'sfdp', 'circo'

root
[Node from G or None (default: None)] The node of G from which to start some layout algorithms.

Returns

Dictionary of (x, y) positions keyed by node.

Notes

This is a wrapper for pydot_layout.

Examples

```
>>> G = nx.complete_graph(4)
>>> pos = nx.nx_pydot.graphviz_layout(G)
>>> pos = nx.nx_pydot.graphviz_layout(G, prog="dot")
```

10.3.8 pydot_layout

pydot_layout(*G*, *prog='neato'*, *root=None*)

Create node positions using `pydot` and Graphviz.

Parameters

G
[Graph] NetworkX graph to be laid out.

prog
[string (default: 'neato')] Name of the GraphViz command to use for layout. Options depend on GraphViz version but may include: 'dot', 'twopi', 'fdp', 'sfdp', 'circo'

root
[Node from G or None (default: None)] The node of G from which to start some layout algorithms.

Returns

dict
Dictionary of positions keyed by node.

Notes

If you use complex node objects, they may have the same string representation and GraphViz could treat them as the same node. The layout may assign both nodes a single location. See Issue #1568 If this occurs in your case, consider relabeling the nodes just for the layout computation using something similar to:

```
H = nx.convert_node_labels_to_integers(G, label_attribute='node_label')
H_layout = nx.nx_pydot.pydot_layout(G, prog='dot')
G_layout = {H.nodes[n]['node_label']: p for n, p in H_layout.items()}
```

Examples

```
>>> G = nx.complete_graph(4)
>>> pos = nx.nx_pydot.pydot_layout(G)
>>> pos = nx.nx_pydot.pydot_layout(G, prog="dot")
```

10.4 Graph Layout

Node positioning algorithms for graph drawing.

For *random_layout()* the possible resulting shape is a square of side [0, scale] (default: [0, 1]) Changing `center` shifts the layout by that amount.

For the other layout routines, the extent is [center - scale, center + scale] (default: [-1, 1]).

Warning: Most layout routines have only been tested in 2-dimensions.

bipartite_layout(G, nodes[, align, scale, ...])	Position nodes in two straight lines.
circular_layout(G[, scale, center, dim])	Position nodes on a circle.
kamada_kawai_layout(G[, dist, pos, weight, ...])	Position nodes using Kamada-Kawai path-length cost-function.
planar_layout(G[, scale, center, dim])	Position nodes without edge intersections.
random_layout(G[, center, dim, seed])	Position nodes uniformly at random in the unit square.
rescale_layout(pos[, scale])	Returns scaled position array to (-scale, scale) in all axes.
rescale_layout_dict(pos[, scale])	Return a dictionary of scaled positions keyed by node
shell_layout(G[, nlist, rotate, scale, ...])	Position nodes in concentric circles.
spring_layout(G[, k, pos, fixed, ...])	Position nodes using Fruchterman-Reingold force-directed algorithm.
spectral_layout(G[, weight, scale, center, dim])	Position nodes using the eigenvectors of the graph Laplacian.
spiral_layout(G[, scale, center, dim, ...])	Position nodes in a spiral layout.
multipartite_layout(G[, subset_key, align, ...])	Position nodes in layers of straight lines.

10.4.1 bipartite_layout

bipartite_layout(*G*, *nodes*, *align='vertical'*, *scale=1*, *center=None*, *aspect_ratio=1.3333333333333333*)

Position nodes in two straight lines.

Parameters

G
[NetworkX graph or list of nodes] A position will be assigned to every node in G.

nodes
[list or container] Nodes in one node set of the bipartite graph. This set will be placed on left or top.

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.

aspect_ratio
[number (default=4/3):] The ratio of the width to the height of the layout.

Returns

pos
[dict] A dictionary of positions keyed by node.

Notes

This algorithm currently only works in two dimensions and does not try to minimize edge crossings.

Examples

```
>>> G = nx.bipartite.gnmk_random_graph(3, 5, 10, seed=123)
>>> top = nx.bipartite.sets(G)[0]
>>> pos = nx.bipartite_layout(G, top)
```

10.4.2 circular_layout

circular_layout(*G*, *scale=1*, *center=None*, *dim=2*)

Position nodes on a circle.

Parameters

G
[NetworkX graph or list of nodes] A position will be assigned to every node in G.

scale
[number (default: 1)] Scale factor for positions.

center
[array-like or None] Coordinate pair around which to center the layout.

dim
[int] Dimension of layout. If dim>2, the remaining dimensions are set to zero in the returned positions. If dim<2, a ValueError is raised.

Returns

pos
[dict] A dictionary of positions keyed by node

Raises

ValueError
If dim < 2

Notes

This algorithm currently only works in two dimensions and does not try to minimize edge crossings.

Examples

```
>>> G = nx.path_graph(4)
>>> pos = nx.circular_layout(G)
```

10.4.3 kamada_kawai_layout

kamada_kawai_layout(*G*, *dist=None*, *pos=None*, *weight='weight'*, *scale=1*, *center=None*, *dim=2*)

Position nodes using Kamada-Kawai path-length cost-function.

Parameters

G
[NetworkX graph or list of nodes] A position will be assigned to every node in G.

dist
[dict (default=None)] A two-level dictionary of optimal distances between nodes, indexed by source and destination node. If None, the distance is computed using shortest_path_length().

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 circular_layout() for dim >= 2 and a linear layout for dim == 1.

weight
[string or None optional (default='weight')] The edge attribute that holds the numerical value used for the edge weight. If None, then all edge weights are 1.

scale
[number (default: 1)] Scale factor for positions.

center
[array-like or None] Coordinate pair around which to center the layout.

dim
[int] Dimension of layout.

Returns

pos
[dict] A dictionary of positions keyed by node

Examples

```
>>> G = nx.path_graph(4)
>>> pos = nx.kamada_kawai_layout(G)
```

10.4.4 planar_layout

planar_layout(*G*, *scale=1*, *center=None*, *dim=2*)

Position nodes without edge intersections.

Parameters

G
[NetworkX graph or list of nodes] A position will be assigned to every node in G. If G is of type nx.PlanarEmbedding, the positions are selected accordingly.

scale
[number (default: 1)] Scale factor for positions.

center
[array-like or None] Coordinate pair around which to center the layout.

dim
[int] Dimension of layout.

Returns

pos
[dict] A dictionary of positions keyed by node

Raises

NetworkXException
If G is not planar

Examples

```
>>> G = nx.path_graph(4)
>>> pos = nx.planar_layout(G)
```

10.4.5 random_layout

random_layout(*G*, *center=None*, *dim=2*, *seed=None*)

Position nodes uniformly at random in the unit square.

For every node, a position is generated by choosing each of dim coordinates uniformly at random on the interval [0.0, 1.0).

NumPy (http://scipy.org) is required for this function.

Parameters

G
[NetworkX graph or list of nodes] A position will be assigned to every node in G.

center
[array-like or None] Coordinate pair around which to center the layout.

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.lollipop_graph(4, 3)
>>> pos = nx.random_layout(G)
```

10.4.6 rescale_layout

rescale_layout(*pos*, *scale=1*)

Returns scaled position array to (-scale, scale) in all axes.

The function acts on NumPy arrays which hold position information. Each position is one row of the array. The dimension of the space equals the number of columns. Each coordinate in one column.

To rescale, the mean (center) is subtracted from each axis separately. Then all values are scaled so that the largest magnitude value from all axes equals `scale` (thus, the aspect ratio is preserved). The resulting NumPy Array is returned (order of rows unchanged).

Parameters

pos
[numpy array] positions to be scaled. Each row is a position.

scale
[number (default: 1)] The size of the resulting extent in all directions.

Returns

pos
[numpy array] scaled positions. Each row is a position.

See also:

rescale_layout_dict

10.4.7 rescale_layout_dict

rescale_layout_dict(*pos*, *scale=1*)

Return a dictionary of scaled positions keyed by node

Parameters

pos
[A dictionary of positions keyed by node]

scale
[number (default: 1)] The size of the resulting extent in all directions.

Returns

pos
[A dictionary of positions keyed by node]

See also:

rescale_layout

Examples

```
>>> import numpy as np
>>> pos = {0: np.array((0, 0)), 1: np.array((1, 1)), 2: np.array((0.5, 0.5))}
>>> nx.rescale_layout_dict(pos)
{0: array([-1., -1.]), 1: array([1., 1.]), 2: array([0., 0.])}
```

```
>>> pos = {0: np.array((0, 0)), 1: np.array((-1, 1)), 2: np.array((-0.5, 0.5))}
>>> nx.rescale_layout_dict(pos, scale=2)
{0: array([ 2., -2.]), 1: array([-2.,  2.]), 2: array([0., 0.])}
```

10.4.8 shell_layout

shell_layout(*G*, *nlist=None*, *rotate=None*, *scale=1*, *center=None*, *dim=2*)

Position nodes in concentric circles.

Parameters

G
[NetworkX graph or list of nodes] A position will be assigned to every node in G.

nlist
[list of lists] List of node lists for each shell.

rotate
[angle in radians (default=pi/len(nlist))] Angle by which to rotate the starting position of each shell relative to the starting position of the previous shell. To recreate behavior before v2.5 use rotate=0.

scale
[number (default: 1)] Scale factor for positions.

center
[array-like or None] Coordinate pair around which to center the layout.

dim
[int] Dimension of layout, currently only dim=2 is supported. Other dimension values result in a ValueError.

Returns

pos
[dict] A dictionary of positions keyed by node

Raises

ValueError
If dim != 2

Notes

This algorithm currently only works in two dimensions and does not try to minimize edge crossings.

Examples

```
>>> G = nx.path_graph(4)
>>> shells = [[0], [1, 2, 3]]
>>> pos = nx.shell_layout(G, shells)
```

10.4.9 spring_layout

spring_layout(*G*, *k=None*, *pos=None*, *fixed=None*, *iterations=50*, *threshold=0.0001*, *weight='weight'*, *scale=1*, *center=None*, *dim=2*, *seed=None*)

Position nodes using Fruchterman-Reingold force-directed algorithm.

The algorithm simulates a force-directed representation of the network treating edges as springs holding nodes close, while treating nodes as repelling objects, sometimes called an anti-gravity force. Simulation continues until the positions are close to an equilibrium.

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)
```

The same using longer but equivalent function name >>> pos = nx.fruchterman_reingold_layout(G)

10.4.10 spectral_layout

spectral_layout(*G*, *weight='weight'*, *scale=1*, *center=None*, *dim=2*)

Position nodes using the eigenvectors of the graph Laplacian.

Using the unnormalized Laplacian, the layout shows possible clusters of nodes which are an approximation of the ratio cut. If dim is the number of dimensions then the positions are the entries of the dim eigenvectors corresponding to the ascending eigenvalues starting from the second one.

Parameters

G
[NetworkX graph or list of nodes] A position will be assigned to every node in G.

weight
[string or None optional (default='weight')] The edge attribute that holds the numerical value used for the edge weight. If None, then all edge weights are 1.

scale
[number (default: 1)] Scale factor for positions.

center
[array-like or None] Coordinate pair around which to center the layout.

dim
[int] Dimension of layout.

Returns

pos
[dict] A dictionary of positions keyed by node

Notes

Directed graphs will be considered as undirected graphs when positioning the nodes.

For larger graphs (>500 nodes) this will use the SciPy sparse eigenvalue solver (ARPACK).

Examples

```
>>> G = nx.path_graph(4)
>>> pos = nx.spectral_layout(G)
```

10.4.11 spiral_layout

spiral_layout(*G*, *scale=1*, *center=None*, *dim=2*, *resolution=0.35*, *equidistant=False*)

Position nodes in a spiral layout.

Parameters

G
[NetworkX graph or list of nodes] A position will be assigned to every node in G.

scale
[number (default: 1)] Scale factor for positions.

center
[array-like or None] Coordinate pair around which to center the layout.

dim
[int, default=2] Dimension of layout, currently only dim=2 is supported. Other dimension values result in a ValueError.

resolution
[float, default=0.35] The compactness of the spiral layout returned. Lower values result in more compressed spiral layouts.

equidistant
[bool, default=False] If True, nodes will be positioned equidistant from each other by decreasing angle further from center. If False, nodes will be positioned at equal angles from each other by increasing separation further from center.

Returns

pos
[dict] A dictionary of positions keyed by node

Raises

ValueError
If dim != 2

Notes

This algorithm currently only works in two dimensions.

Examples

```
>>> G = nx.path_graph(4)
>>> pos = nx.spiral_layout(G)
>>> nx.draw(G, pos=pos)
```

10.4.12 multipartite_layout

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 (default='subset')] Key of node data to be used as 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.

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.

Examples

```
>>> G = nx.complete_multipartite_graph(28, 16, 10)
>>> pos = nx.multipartite_layout(G)
```

CHAPTER

ELEVEN

RANDOMNESS

Random Number Generators (RNGs) are often used when generating, drawing and computing properties or manipulating networks. NetworkX provides functions which use one of two standard RNGs: NumPy's package `numpy.random` or Python's built-in package `random`. They each provide the same algorithm for generating numbers (Mersenne Twister). Their interfaces are similar (dangerously similar) and yet distinct. They each provide a global default instance of their generator that is shared by all programs in a single session. For the most part you can use the RNGs as NetworkX has them set up and you'll get reasonable pseudorandom results (results that are statistically random, but created in a deterministic manner).

Sometimes you want more control over how the numbers are generated. In particular, you need to set the `seed` of the generator to make your results reproducible – either for scientific publication or for debugging. Both RNG packages have easy functions to set the seed to any integer, thus determining the subsequent generated values. Since this package (and many others) use both RNGs you may need to set the `seed` of both RNGs. Even if we strictly only used one of the RNGs, you may find yourself using another package that uses the other. Setting the state of the two global RNGs is as simple setting the seed of each RNG to an arbitrary integer:

```
>>> import random
>>> random.seed(246)         # or any integer
>>> import numpy
>>> numpy.random.seed(4812)
```

Many users will be satisfied with this level of control.

For people who want even more control, we include an optional argument to functions that use an RNG. This argument is called `seed`, but determines more than the seed of the RNG. It tells the function which RNG package to use, and whether to use a global or local RNG.

```
>>> from networkx import path_graph, random_layout
>>> G = path_graph(9)
>>> pos = random_layout(G, seed=None)  # use (either) global default RNG
>>> pos = random_layout(G, seed=42)  # local RNG just for this call
>>> pos = random_layout(G, seed=numpy.random)  # use numpy global RNG
>>> random_state = numpy.random.RandomState(42)
>>> pos = random_layout(G, seed=random_state)  # use/reuse your own RNG
```

Each NetworkX function that uses an RNG was written with one RNG package in mind. It either uses `random` or `numpy.random` by default. But some users want to only use a single RNG for all their code. This `seed` argument provides a mechanism so that any function can use a `numpy.random` RNG even if the function is written for `random`. It works as follows.

The default behavior (when `seed=None`) is to use the global RNG for the function's preferred package. If seed is set to an integer value, a local RNG is created with the indicated seed value and is used for the duration of that function (including any calls to other functions) and then discarded. Alternatively, you can specify `seed=numpy.random` to ensure that the global numpy RNG is used whether the function expects it or not. Finally, you can provide a numpy RNG

to be used by the function. The RNG is then available to use in other functions or even other package like sklearn. In this way you can use a single RNG for all random numbers in your project.

While it is possible to assign `seed` a `random`-style RNG for NetworkX functions written for the `random` package API, the numpy RNG interface has too many nice features for us to ensure a `random`-style RNG will work in all functions. In practice, you can do most things using only `random` RNGs (useful if numpy is not available). But your experience will be richer if numpy is available.

To summarize, you can easily ignore the `seed` argument and use the global RNGs. You can specify to use only the numpy global RNG with `seed=numpy.random`. You can use a local RNG by providing an integer seed value. And you can provide your own numpy RNG, reusing it for all functions. It is easier to use numpy RNGs if you want a single RNG for your computations.

CHAPTER

TWELVE

EXCEPTIONS

Base exceptions and errors for NetworkX.

class NetworkXException

Base class for exceptions in NetworkX.

class NetworkXError

Exception for a serious error in NetworkX

class NetworkXPointlessConcept

Raised when a null graph is provided as input to an algorithm that cannot use it.

The null graph is sometimes considered a pointless concept [1], thus the name of the exception.

References

[1]

class NetworkXAlgorithmError

Exception for unexpected termination of algorithms.

class NetworkXUnfeasible

Exception raised by algorithms trying to solve a problem instance that has no feasible solution.

class NetworkXNoPath

Exception for algorithms that should return a path when running on graphs where such a path does not exist.

class NetworkXNoCycle

Exception for algorithms that should return a cycle when running on graphs where such a cycle does not exist.

class NodeNotFound

Exception raised if requested node is not present in the graph

class HasACycle

Raised if a graph has a cycle when an algorithm expects that it will have no cycles.

class NetworkXUnbounded

Exception raised by algorithms trying to solve a maximization or a minimization problem instance that is unbounded.

class NetworkXNotImplemented

Exception raised by algorithms not implemented for a type of graph.

class AmbiguousSolution

Raised if more than one valid solution exists for an intermediary step of an algorithm.

In the face of ambiguity, refuse the temptation to guess. This may occur, for example, when trying to determine the bipartite node sets in a disconnected bipartite graph when computing bipartite matchings.

class ExceededMaxIterations

Raised if a loop iterates too many times without breaking.

This may occur, for example, in an algorithm that computes progressively better approximations to a value but exceeds an iteration bound specified by the user.

class PowerIterationFailedConvergence(*num_iterations*, **args*, ***kw*)

Raised when the power iteration method fails to converge within a specified iteration limit.

`num_iterations` is the number of iterations that have been completed when this exception was raised.

CHAPTER

THIRTEEN

UTILITIES

13.1 Helper Functions

Miscellaneous Helpers for NetworkX.

These are not imported into the base networkx namespace but can be accessed, for example, as

```
>>> import networkx
>>> networkx.utils.make_list_of_ints({1, 2, 3})
[1, 2, 3]
>>> networkx.utils.arbitrary_element({5, 1, 7})
1
```

arbitrary_element(iterable)	Returns an arbitrary element of `iterable` without removing it.
flatten(obj[, result])	Return flattened version of (possibly nested) iterable object.
make_list_of_ints(sequence)	Return list of ints from sequence of integral numbers.
dict_to_numpy_array(d[, mapping])	Convert a dictionary of dictionaries to a numpy array with optional mapping.
pairwise(iterable[, cyclic])	s -> (s0, s1), (s1, s2), (s2, s3), ...
groups(many_to_one)	Converts a many-to-one mapping into a one-to-many mapping.
create_random_state([random_state])	Returns a numpy.random.RandomState or numpy.random.Generator instance depending on input.
create_py_random_state([random_state])	Returns a random.Random instance depending on input.
nodes_equal(nodes1, nodes2)	Check if nodes are equal.
edges_equal(edges1, edges2)	Check if edges are equal.
graphs_equal(graph1, graph2)	Check if graphs are equal.

13.1.1 arbitrary_element

arbitrary_element(*iterable*)

Returns an arbitrary element of `iterable` without removing it.

This is most useful for "peeking" at an arbitrary element of a set, but can be used for any list, dictionary, etc., as well.

Parameters

iterable
[`abc.collections.Iterable` instance] Any object that implements `__iter__`, e.g. set, dict, list, tuple, etc.

Returns

The object that results from `next(iter(iterable))`

Raises

ValueError
If `iterable` is an iterator (because the current implementation of this function would consume an element from the iterator).

Notes

This function does not return a *random* element. If `iterable` is ordered, sequential calls will return the same value:

```
>>> l = [1, 2, 3]
>>> nx.utils.arbitrary_element(l)
1
>>> nx.utils.arbitrary_element(l)
1
```

Examples

Arbitrary elements from common Iterable objects:

```
>>> nx.utils.arbitrary_element([1, 2, 3])  # list
1
>>> nx.utils.arbitrary_element((1, 2, 3))  # tuple
1
>>> nx.utils.arbitrary_element({1, 2, 3})  # set
1
>>> d = {k: v for k, v in zip([1, 2, 3], [3, 2, 1])}
>>> nx.utils.arbitrary_element(d)  # dict_keys
1
>>> nx.utils.arbitrary_element(d.values())  # dict values
3
```

`str` is also an Iterable:

```
>>> nx.utils.arbitrary_element("hello")
'h'
```

`ValueError` is raised if `iterable` is an iterator:

```
>>> iterator = iter([1, 2, 3])  # Iterator, *not* Iterable
>>> nx.utils.arbitrary_element(iterator)
Traceback (most recent call last):
    ...
ValueError: cannot return an arbitrary item from an iterator
```

13.1.2 flatten

flatten(*obj*, *result=None*)

Return flattened version of (possibly nested) iterable object.

13.1.3 make_list_of_ints

make_list_of_ints(*sequence*)

Return list of ints from sequence of integral numbers.

All elements of the sequence must satisfy int(element) == element or a ValueError is raised. Sequence is iterated through once.

If sequence is a list, the non-int values are replaced with ints. So, no new list is created

13.1.4 dict_to_numpy_array

dict_to_numpy_array(*d*, *mapping=None*)

Convert a dictionary of dictionaries to a numpy array with optional mapping.

13.1.5 pairwise

pairwise(*iterable*, *cyclic=False*)

s -> (s0, s1), (s1, s2), (s2, s3), …

13.1.6 groups

groups(*many_to_one*)

Converts a many-to-one mapping into a one-to-many mapping.

`many_to_one` must be a dictionary whose keys and values are all hashable.

The return value is a dictionary mapping values from `many_to_one` to sets of keys from `many_to_one` that have that value.

Examples

```
>>> from networkx.utils import groups
>>> many_to_one = {"a": 1, "b": 1, "c": 2, "d": 3, "e": 3}
>>> groups(many_to_one)
{1: {'a', 'b'}, 2: {'c'}, 3: {'e', 'd'}}
```

13.1.7 create_random_state

create_random_state(*random_state=None*)

Returns a numpy.random.RandomState or numpy.random.Generator instance depending on input.

Parameters

random_state
[int or NumPy RandomState or Generator instance, optional (default=None)] If int, return a numpy.random.RandomState instance set with seed=int. if `numpy.random.RandomState` instance, return it. if `numpy.random.Generator` instance, return it. if None or numpy.random, return the global random number generator used by numpy.random.

13.1.8 create_py_random_state

create_py_random_state(*random_state=None*)

Returns a random.Random instance depending on input.

Parameters

random_state
[int or random number generator or None (default=None)] If int, return a random.Random instance set with seed=int. if random.Random instance, return it. if None or the `random` package, return the global random number generator used by `random`. if np.random package, return the global numpy random number generator wrapped in a PythonRandomInterface class. if np.random.RandomState or np.random.Generator instance, return it wrapped in PythonRandomInterface if a PythonRandomInterface instance, return it

13.1.9 nodes_equal

nodes_equal(*nodes1*, *nodes2*)

Check if nodes are equal.

Equality here means equal as Python objects. Node data must match if included. The order of nodes is not relevant.

Parameters

nodes1, nodes2
[iterables of nodes, or (node, datadict) tuples]

Returns

bool
True if nodes are equal, False otherwise.

13.1.10 edges_equal

edges_equal(*edges1*, *edges2*)

Check if edges are equal.

Equality here means equal as Python objects. Edge data must match if included. The order of the edges is not relevant.

Parameters

edges1, edges2
[iterables of with u, v nodes as] edge tuples (u, v), or edge tuples with data dicts (u, v, d), or edge tuples with keys and data dicts (u, v, k, d)

Returns

bool
True if edges are equal, False otherwise.

13.1.11 graphs_equal

graphs_equal(*graph1*, *graph2*)

Check if graphs are equal.

Equality here means equal as Python objects (not isomorphism). Node, edge and graph data must match.

Parameters

graph1, graph2
[graph]

Returns

bool
True if graphs are equal, False otherwise.

13.2 Data Structures and Algorithms

Union-find data structure.

UnionFind.union(*objects)	Find the sets containing the objects and merge them all.

13.2.1 UnionFind.union

UnionFind.**union**(**objects*)

Find the sets containing the objects and merge them all.

13.3 Random Sequence Generators

Utilities for generating random numbers, random sequences, and random selections.

powerlaw_sequence(n[, exponent, seed])	Return sample sequence of length n from a power law distribution.
cumulative_distribution(distribution)	Returns normalized cumulative distribution from discrete distribution.
discrete_sequence(n[, distribution, ...])	Return sample sequence of length n from a given discrete distribution or discrete cumulative distribution.
zipf_rv(alpha[, xmin, seed])	Returns a random value chosen from the Zipf distribution.
random_weighted_sample(mapping, k[, seed])	Returns k items without replacement from a weighted sample.
weighted_choice(mapping[, seed])	Returns a single element from a weighted sample.

13.3.1 powerlaw_sequence

powerlaw_sequence(*n*, *exponent=2.0*, *seed=None*)

Return sample sequence of length n from a power law distribution.

13.3.2 cumulative_distribution

cumulative_distribution(*distribution*)

Returns normalized cumulative distribution from discrete distribution.

13.3.3 discrete_sequence

discrete_sequence(*n*, *distribution=None*, *cdistribution=None*, *seed=None*)

Return sample sequence of length n from a given discrete distribution or discrete cumulative distribution.

One of the following must be specified.

distribution = histogram of values, will be normalized

cdistribution = normalized discrete cumulative distribution

13.3.4 zipf_rv

zipf_rv(*alpha*, *xmin=1*, *seed=None*)

Returns a random value chosen from the Zipf distribution.

The return value is an integer drawn from the probability distribution

$$p(x) = \frac{x^{-\alpha}}{\zeta(\alpha, x_{\min})},$$

where $\zeta(\alpha, x_{\min})$ is the Hurwitz zeta function.

Parameters

alpha
[float] Exponent value of the distribution

xmin
[int] Minimum value

seed
[integer, random_state, or None (default)] Indicator of random number generation state. See *Randomness*.

Returns

x
[int] Random value from Zipf distribution

Raises

ValueError:
If xmin < 1 or If alpha <= 1

Notes

The rejection algorithm generates random values for a the power-law distribution in uniformly bounded expected time dependent on parameters. See [1] for details on its operation.

References

[1]

Examples

```
>>> nx.utils.zipf_rv(alpha=2, xmin=3, seed=42)
8
```

13.3.5 random_weighted_sample

random_weighted_sample(*mapping*, *k*, *seed=None*)

Returns k items without replacement from a weighted sample.

The input is a dictionary of items with weights as values.

13.3.6 weighted_choice

weighted_choice(*mapping*, *seed=None*)

Returns a single element from a weighted sample.

The input is a dictionary of items with weights as values.

13.4 Decorators

open_file(path_arg[, mode])	Decorator to ensure clean opening and closing of files.
not_implemented_for(*graph_types)	Decorator to mark algorithms as not implemented
nodes_or_number(which_args)	Decorator to allow number of nodes or container of nodes.
np_random_state(random_state_argument)	Decorator to generate a `numpy.random.RandomState` instance.
py_random_state(random_state_argument)	Decorator to generate a random.Random instance (or equiv).
argmap(func, *args[, try_finally])	A decorator to apply a map to arguments before calling the function

13.4.1 open_file

open_file(*path_arg*, *mode='r'*)

Decorator to ensure clean opening and closing of files.

Parameters

path_arg
[string or int] Name or index of the argument that is a path.

mode
[str] String for opening mode.

Returns

_open_file
[function] Function which cleanly executes the io.

Notes

Note that this decorator solves the problem when a path argument is specified as a string, but it does not handle the situation when the function wants to accept a default of None (and then handle it).

Here is an example of how to handle this case:

```
@open_file("path")
def some_function(arg1, arg2, path=None):
    if path is None:
        fobj = tempfile.NamedTemporaryFile(delete=False)
    else:
        # `path` could have been a string or file object or something
        # similar. In any event, the decorator has given us a file object
        # and it will close it for us, if it should.
        fobj = path

    try:
        fobj.write("blah")
    finally:
        if path is None:
            fobj.close()
```

Normally, we'd want to use "with" to ensure that fobj gets closed. However, the decorator will make `path` a file object for us, and using "with" would undesirably close that file object. Instead, we use a try block, as shown above. When we exit the function, fobj will be closed, if it should be, by the decorator.

Examples

Decorate functions like this:

```
@open_file(0,"r")
def read_function(pathname):
    pass

@open_file(1,"w")
def write_function(G, pathname):
    pass

@open_file(1,"w")
def write_function(G, pathname="graph.dot"):
    pass

@open_file("pathname","w")
def write_function(G, pathname="graph.dot"):
    pass

@open_file("path", "w+")
def another_function(arg, **kwargs):
    path = kwargs["path"]
    pass
```

13.4.2 not_implemented_for

not_implemented_for(**graph_types*)

Decorator to mark algorithms as not implemented

Parameters

graph_types
[container of strings] Entries must be one of "directed", "undirected", "multigraph", or "graph".

Returns

_require
[function] The decorated function.

Raises

NetworkXNotImplemented
If any of the packages cannot be imported

Notes

Multiple types are joined logically with "and". For "or" use multiple @not_implemented_for() lines.

Examples

Decorate functions like this:

```
@not_implemented_for("directed")
def sp_function(G):
    pass

# rule out MultiDiGraph
@not_implemented_for("directed","multigraph")
def sp_np_function(G):
    pass

# rule out all except DiGraph
@not_implemented_for("undirected")
@not_implemented_for("multigraph")
def sp_np_function(G):
    pass
```

13.4.3 nodes_or_number

nodes_or_number(*which_args*)

Decorator to allow number of nodes or container of nodes.

With this decorator, the specified argument can be either a number or a container of nodes. If it is a number, the nodes used are `range(n)`. This allows `nx.complete_graph(50)` in place of `nx.complete_graph(list(range(50)))`. And it also allows `nx.complete_graph(any_list_of_nodes)`.

Parameters

which_args
[string or int or sequence of strings or ints] If string, the name of the argument to be treated. If int, the index of the argument to be treated. If more than one node argument is allowed, can be a list of locations.

Returns

_nodes_or_numbers
[function] Function which replaces int args with ranges.

Examples

Decorate functions like this:

```
@nodes_or_number("nodes")
def empty_graph(nodes):
    # nodes is converted to a list of nodes

@nodes_or_number(0)
def empty_graph(nodes):
    # nodes is converted to a list of nodes

@nodes_or_number(["m1", "m2"])
def grid_2d_graph(m1, m2, periodic=False):
    # m1 and m2 are each converted to a list of nodes

@nodes_or_number([0, 1])
def grid_2d_graph(m1, m2, periodic=False):
    # m1 and m2 are each converted to a list of nodes

@nodes_or_number(1)
def full_rary_tree(r, n)
    # presumably r is a number. It is not handled by this decorator.
    # n is converted to a list of nodes
```

13.4.4 np_random_state

np_random_state(*random_state_argument*)

Decorator to generate a `numpy.random.RandomState` instance.

The decorator processes the argument indicated by `random_state_argument` using `nx.utils.create_random_state()`. The argument value can be a seed (integer), or a `numpy.random.RandomState` instance or (`None` or `numpy.random`). The latter options use the glocal random number generator used by `numpy.random`. The result is a `numpy.random.RandomState` instance.

Parameters

random_state_argument
: [string or int] The name or index of the argument to be converted to a `numpy.random.RandomState` instance.

Returns

_random_state
: [function] Function whose random_state keyword argument is a RandomState instance.

See also:

py_random_state

Examples

Decorate functions like this:

```
@np_random_state("seed")
def random_float(seed=None):
    return seed.rand()

@np_random_state(0)
def random_float(rng=None):
    return rng.rand()

@np_random_state(1)
def random_array(dims, random_state=1):
    return random_state.rand(*dims)
```

13.4.5 py_random_state

py_random_state(*random_state_argument*)

Decorator to generate a random.Random instance (or equiv).

The decorator processes the argument indicated by `random_state_argument` using `nx.utils.create_py_random_state()`. The argument value can be a seed (integer), or a random number generator:

```
If int, return a random.Random instance set with seed=int.
If random.Random instance, return it.
If None or the `random` package, return the global random number
generator used by `random`.
If np.random package, return the global numpy random number
generator wrapped in a PythonRandomInterface class.
If np.random.RandomState instance, return it wrapped in
PythonRandomInterface
If a PythonRandomInterface instance, return it
```

Parameters

random_state_argument
: [string or int] The name of the argument or the index of the argument in args that is to be converted to the random.Random instance or numpy.random.RandomState instance that mimics basic methods of random.Random.

Returns

_random_state
: [function] Function whose random_state_argument is converted to a Random instance.

See also:

np_random_state

Examples

Decorate functions like this:

```
@py_random_state("random_state")
def random_float(random_state=None):
    return random_state.rand()

@py_random_state(0)
def random_float(rng=None):
    return rng.rand()

@py_random_state(1)
def random_array(dims, seed=12345):
    return seed.rand(*dims)
```

13.4.6 networkx.utils.decorators.argmap

class argmap(*func*, **args*, *try_finally=False*)

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.

See also:

not_implemented_for
open_file
nodes_or_number
random_state
py_random_state
networkx.community.quality.require_partition
require_partition

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.

3) The maps applied can process multiple arguments. For example, you could swap two arguments using a mapping, or transform them to their sum and their difference. This was included to allow a decorator in the `quality.py` module that checks that an input `partition` is a valid partition of the nodes of the input graph `G`. In this example, the map has inputs `(G, partition)`. After checking for a valid partition, the map either raises an exception or leaves the inputs unchanged. Thus many functions that make this check can use the decorator rather than copy the checking code into each function. More complicated nested argument structures are described below.

The remaining notes describe the code structure and methods for this class in broad terms to aid in understanding how to use it.

Instantiating an *argmap* object simply stores the mapping function and the input identifiers of which arguments to map. The resulting decorator is ready to use this map to decorate any function. Calling that object (`argmap.__call__`, but usually done via `@my_decorator`) a lazily compiled thin wrapper of the decorated function is constructed, wrapped with the necessary function dunder attributes like `__doc__` and `__name__`. That thinly wrapped function is returned as the decorated function. When that decorated function is called, the thin wrapper of code calls `argmap._lazy_compile` which compiles the decorated function (using *argmap.compile*) and replaces the code of the thin wrapper with the newly compiled code. This saves the compilation step every import of networkx, at the cost of compiling upon the first call to the decorated function.

When the decorated function is compiled, the code is recursively assembled using the *argmap.assemble* method. The recursive nature is needed in case of nested decorators. The result of the assembly is a number of useful objects.

sig
: [the function signature of the original decorated function as] constructed by *argmap.signature()*. This is constructed using `inspect.signature` but enhanced with attribute strings `sig_def` and `sig_call`, and other information specific to mapping arguments of this function. This information is used to construct a string of code defining the new decorated function.

wrapped_name
: [a unique internally used name constructed by argmap] for the decorated function.

functions
: [a dict of the functions used inside the code of this] decorated function, to be used as `globals` in `exec`. This dict is recursively updated to allow for nested decorating.

mapblock
: [code (as a list of strings) to map the incoming argument] values to their mapped values.

finallys
: [code (as a list of strings) to provide the possibly nested] set of finally clauses if needed.

mutable_args
: [a bool indicating whether the `sig.args` tuple should be] converted to a list so mutation can occur.

After this recursive assembly process, the *`argmap.compile`* method constructs code (as strings) to convert the tuple `sig.args` to a list if needed. It joins the defining code with appropriate indents and compiles the result. Finally, this code is evaluated and the original wrapper's implementation is replaced with the compiled version (see `argmap._lazy_compile` for more details).

Other *`argmap`* methods include `_name` and `_count` which allow internally generated names to be unique within a python session. The methods `_flatten` and `_indent` process the nested lists of strings into properly indented python code ready to be compiled.

More complicated nested tuples of arguments also allowed though usually not used. For the simple 2 argument case, the argmap input ("a", "b") implies the mapping function will take 2 arguments and return a 2-tuple of mapped values. A more complicated example with argmap input `("a", ("b", "c"))` requires the mapping function take 2 inputs, with the second being a 2-tuple. It then must output the 3 mapped values in the same nested structure `(newa, (newb, newc))`. This level of generality is not often needed, but was convenient to implement when handling the multiple arguments.

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):
    return xlist - y + zlist
```

or:

```
@argmap(sum, "zlist", 0)
def foo(xlist, y, zlist):
    return xlist - y + zlist
```

Transforming functions can be applied to multiple arguments, such as:

```
def swap(x, y):
    return y, x

# the 2-tuple tells argmap that the map `swap` has 2 inputs/outputs.
@argmap(swap, ("a", "b")):
def foo(a, b, c):
    return a / b * c
```

is equivalent to:

```
def foo(a, b, c):
    a, b = swap(a, b)
    return a / b * c
```

More generally, the applied arguments can be nested tuples of strings or ints. The syntax `@argmap(some_func, ("a", ("b", "c")))` would expect `some_func` to accept 2 inputs with the second expected to be a 2-tuple. It should then return 2 outputs with the second a 2-tuple. The returns values would replace input "a" "b" and "c" respectively. Similarly for `@argmap(some_func, (0, ("b", 2)))`.

Also, note that an index larger than the number of named parameters is allowed for variadic functions. For example:

```
def double(a):
    return 2 * a

@argmap(double, 3)
def overflow(a, *args):
    return a, args

print(overflow(1, 2, 3, 4, 5, 6))  # output is 1, (2, 3, 8, 5, 6)
```

Try Finally

Additionally, this *argmap* class can be used to create a decorator that initiates a try…finally block. The decorator must be written to return both the transformed argument and a closing function. This feature was included to enable the *open_file* decorator which might need to close the file or not depending on whether it had to open that file. This feature uses the keyword-only `try_finally` argument to `@argmap`.

For example this map opens a file and then makes sure it is closed:

```
def open_file(fn):
    f = open(fn)
    return f, lambda: f.close()
```

The decorator applies that to the function `foo`:

```
@argmap(open_file, "file", try_finally=True)
def foo(file):
    print(file.read())
```

is syntactic sugar for:

```
def foo(file):
    file, close_file = open_file(file)
    try:
        print(file.read())
    finally:
        close_file()
```

and is equivalent to (using indexes):

```
@argmap(open_file, 0, try_finally=True)
def foo(file):
    print(file.read())
```

Here's an example of the try_finally feature used to create a decorator:

```
def my_closing_decorator(which_arg):
    def _opener(path):
        if path is None:
            path = open(path)
            fclose = path.close
        else:
            # assume `path` handles the closing
            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):
    # this code doesn't need to worry about closing the file
    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):
```

(continues on next page)

(continued from previous page)

```
    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.

__init__(*func*, **args*, *try_finally=False*)

Methods

assemble(f)	Collects components of the source for the decorated function wrapping f.
compile(f)	Compile the decorated function.
signature(f)	Construct a Signature object describing f

argmap.assemble

argmap.**assemble**(*f*)

Collects components of the source for the decorated function wrapping f.

If f has multiple argmap decorators, we recursively assemble the stack of decorators into a single flattened function.

This method is part of the *compile* method's process yet separated from that method to allow recursive processing. The outputs are strings, dictionaries and lists that collect needed info to flatten any nested argmap-decoration.

Parameters

f

[callable] The function to be decorated. If f is argmapped, we assemble it.

Returns

sig
[argmap.Signature] The function signature as an `argmap.Signature` object.

wrapped_name
[str] The mangled name used to represent the wrapped function in the code being assembled.

functions
[dict] A dictionary mapping id(g) -> (mangled_name(g), g) for functions g referred to in the code being assembled. These need to be present in the `globals` scope of `exec` when defining the decorated function.

mapblock
[list of lists and/or strings] Code that implements mapping of parameters including any try blocks if needed. This code will precede the decorated function call.

finallys
[list of lists and/or strings] Code that implements the finally blocks to post-process the arguments (usually close any files if needed) after the decorated function is called.

mutable_args
[bool] True if the decorator needs to modify positional arguments via their indices. The compile method then turns the argument tuple into a list so that the arguments can be modified.

argmap.compile

`argmap.`**`compile`**`(`*`f`*`)`

Compile the decorated function.

Called once for a given decorated function – collects the code from all argmap decorators in the stack, and compiles the decorated function.

Much of the work done here uses the *`assemble`* method to allow recursive treatment of multiple argmap decorators on a single decorated function. That flattens the argmap decorators, collects the source code to construct a single decorated function, then compiles/executes/returns that function.

The source code for the decorated function is stored as an attribute `_code` on the function object itself.

Note that Python's *`compile`* function requires a filename, but this code is constructed without a file, so a fictitious filename is used to describe where the function comes from. The name is something like: "argmap compilation 4".

Parameters

f
[callable] The function to be decorated

Returns

func
[callable] The decorated file

argmap.signature

classmethod argmap.**signature**(*f*)

Construct a Signature object describing `f`

Compute a Signature so that we can write a function wrapping f with the same signature and call-type.

Parameters

f
[callable] A function to be decorated

Returns

sig
[argmap.Signature] The Signature of f

Notes

The Signature is a namedtuple with names:

name : a unique version of the name of the decorated function signature : the inspect.signature of the decorated function def_sig : a string used as code to define the new function call_sig : a string used as code to call the decorated function names : a dict keyed by argument name and index to the argument's name n_positional : the number of positional arguments in the signature args : the name of the VAR_POSITIONAL argument if any, i.e. *theseargs kwargs : the name of the VAR_KEYWORDS argument if any, i.e. **kwargs

These named attributes of the signature are used in *assemble* and *compile* to construct a string of source code for the decorated function.

13.5 Cuthill-Mckee Ordering

Cuthill-McKee ordering of graph nodes to produce sparse matrices

cuthill_mckee_ordering(G[, heuristic])	Generate an ordering (permutation) of the graph nodes to make a sparse matrix.
reverse_cuthill_mckee_ordering(G[, heuristic])	Generate an ordering (permutation) of the graph nodes to make a sparse matrix.

13.5.1 cuthill_mckee_ordering

cuthill_mckee_ordering(*G*, *heuristic=None*)

Generate an ordering (permutation) of the graph nodes to make a sparse matrix.

Uses the Cuthill-McKee heuristic (based on breadth-first search) [1].

Parameters

G
[graph] A NetworkX graph

heuristic
[function, optional] Function to choose starting node for RCM algorithm. If None a node from a pseudo-peripheral pair is used. A user-defined function can be supplied that takes a graph object and returns a single node.

Returns

nodes
[generator] Generator of nodes in Cuthill-McKee ordering.

See also:

reverse_cuthill_mckee_ordering

Notes

The optimal solution the bandwidth reduction is NP-complete [2].

References

[1], [2]

Examples

```
>>> from networkx.utils import cuthill_mckee_ordering
>>> G = nx.path_graph(4)
>>> rcm = list(cuthill_mckee_ordering(G))
>>> A = nx.adjacency_matrix(G, nodelist=rcm)
```

Smallest degree node as heuristic function:

```
>>> def smallest_degree(G):
...     return min(G, key=G.degree)
>>> rcm = list(cuthill_mckee_ordering(G, heuristic=smallest_degree))
```

13.5.2 reverse_cuthill_mckee_ordering

reverse_cuthill_mckee_ordering(*G*, *heuristic=None*)

Generate an ordering (permutation) of the graph nodes to make a sparse matrix.

Uses the reverse Cuthill-McKee heuristic (based on breadth-first search) [1].

Parameters

G
[graph] A NetworkX graph

heuristic
[function, optional] Function to choose starting node for RCM algorithm. If None a node from a pseudo-peripheral pair is used. A user-defined function can be supplied that takes a graph object and returns a single node.

Returns

nodes
[generator] Generator of nodes in reverse Cuthill-McKee ordering.

See also:

cuthill_mckee_ordering

Notes

The optimal solution the bandwidth reduction is NP-complete [2].

References

[1], [2]

Examples

```
>>> from networkx.utils import reverse_cuthill_mckee_ordering
>>> G = nx.path_graph(4)
>>> rcm = list(reverse_cuthill_mckee_ordering(G))
>>> A = nx.adjacency_matrix(G, nodelist=rcm)
```

Smallest degree node as heuristic function:

```
>>> def smallest_degree(G):
...     return min(G, key=G.degree)
>>> rcm = list(reverse_cuthill_mckee_ordering(G, heuristic=smallest_degree))
```

13.6 Mapped Queue

Priority queue class with updatable priorities.

MappedQueue([data])	The MappedQueue class implements a min-heap with removal and update-priority.

13.6.1 networkx.utils.mapped_queue.MappedQueue

class MappedQueue(*data=None*)

The MappedQueue class implements a min-heap with removal and update-priority.

The min heap uses heapq as well as custom written _siftup and _siftdown methods to allow the heap positions to be tracked by an additional dict keyed by element to position. The smallest element can be popped in O(1) time, new elements can be pushed in O(log n) time, and any element can be removed or updated in O(log n) time. The queue cannot contain duplicate elements and an attempt to push an element already in the queue will have no effect.

MappedQueue complements the heapq package from the python standard library. While MappedQueue is designed for maximum compatibility with heapq, it adds element removal, lookup, and priority update.

Parameters

data
[dict or iterable]

References

[1], [2]

Examples

A *MappedQueue* can be created empty, or optionally, given a dictionary of initial elements and priorities. The methods *push*, *pop*, *remove*, and *update* operate on the queue.

```
>>> colors_nm = {'red':665, 'blue': 470, 'green': 550}
>>> q = MappedQueue(colors_nm)
>>> q.remove('red')
>>> q.update('green', 'violet', 400)
>>> q.push('indigo', 425)
True
>>> [q.pop().element for i in range(len(q.heap))]
['violet', 'indigo', 'blue']
```

A *MappedQueue* can also be initialized with a list or other iterable. The priority is assumed to be the sort order of the items in the list.

```
>>> q = MappedQueue([916, 50, 4609, 493, 237])
>>> q.remove(493)
>>> q.update(237, 1117)
>>> [q.pop() for i in range(len(q.heap))]
[50, 916, 1117, 4609]
```

An exception is raised if the elements are not comparable.

```
>>> q = MappedQueue([100, 'a'])
Traceback (most recent call last):
...
TypeError: '<' not supported between instances of 'int' and 'str'
```

To avoid the exception, use a dictionary to assign priorities to the elements.

```
>>> q = MappedQueue({100: 0, 'a': 1 })
```

__init__(*data=None*)
Priority queue class with updatable priorities.

Methods

pop()	Remove and return the smallest element in the queue.
push(elt[, priority])	Add an element to the queue.
remove(elt)	Remove an element from the queue.
update(elt, new[, priority])	Replace an element in the queue with a new one.

MappedQueue.pop

MappedQueue.**pop**()

Remove and return the smallest element in the queue.

MappedQueue.push

MappedQueue.**push**(*elt*, *priority=None*)

Add an element to the queue.

MappedQueue.remove

MappedQueue.**remove**(*elt*)

Remove an element from the queue.

MappedQueue.update

MappedQueue.**update**(*elt*, *new*, *priority=None*)

Replace an element in the queue with a new one.

CHAPTER

FOURTEEN

GLOSSARY

dictionary
: A Python dictionary maps keys to values. Also known as "hashes", or "associative arrays" in other programming languages. See the Python tutorial on dictionaries.

edge
: Edges are either two-tuples of nodes `(u, v)` or three tuples of nodes with an edge attribute dictionary `(u, v, dict)`.

ebunch
: An iteratable container of edge tuples like a list, iterator, or file.

edge attribute
: Edges can have arbitrary Python objects assigned as attributes by using keyword/value pairs when adding an edge assigning to the `G.edges[u][v]` attribute dictionary for the specified edge *u*-*v*.

nbunch
: An nbunch is a single node, container of nodes or `None` (representing all nodes). It can be a list, set, graph, etc.. To filter an nbunch so that only nodes actually in `G` appear, use `G.nbunch_iter(nbunch)`.

node
: A node can be any hashable Python object except None.

node attribute
: Nodes can have arbitrary Python objects assigned as attributes by using keyword/value pairs when adding a node or assigning to the `G.nodes[n]` attribute dictionary for the specified node `n`.

APPENDIX

A

TUTORIAL

This guide can help you start working with NetworkX.

A.1 Creating a graph

Create an empty graph with no nodes and no edges.

```
>>> import networkx as nx
>>> G = nx.Graph()
```

By definition, a *Graph* is a collection of nodes (vertices) along with identified pairs of nodes (called edges, links, etc). In NetworkX, nodes can be any hashable object e.g., a text string, an image, an XML object, another Graph, a customized node object, etc.

Note: Python's `None` object is not allowed to be used as a node. It determines whether optional function arguments have been assigned in many functions.

A.2 Nodes

The graph `G` can be grown in several ways. NetworkX includes many *graph generator functions* and *facilities to read and write graphs in many formats*. To get started though we'll look at simple manipulations. You can add one node at a time,

```
>>> G.add_node(1)
```

or add nodes from any iterable container, such as a list

```
>>> G.add_nodes_from([2, 3])
```

You can also add nodes along with node attributes if your container yields 2-tuples of the form `(node, node_attribute_dict)`:

```
>>> G.add_nodes_from([
...     (4, {"color": "red"}),
...     (5, {"color": "green"}),
... ])
```

Node attributes are discussed further *below*.

Nodes from one graph can be incorporated into another:

```
>>> H = nx.path_graph(10)
>>> G.add_nodes_from(H)
```

G now contains the nodes of H as nodes of G. In contrast, you could use the graph H as a node in G.

```
>>> G.add_node(H)
```

The graph G now contains H as a node. This flexibility is very powerful as it allows graphs of graphs, graphs of files, graphs of functions and much more. It is worth thinking about how to structure your application so that the nodes are useful entities. Of course you can always use a unique identifier in G and have a separate dictionary keyed by identifier to the node information if you prefer.

Note: You should not change the node object if the hash depends on its contents.

A.3 Edges

G can also be grown by adding one edge at a time,

```
>>> G.add_edge(1, 2)
>>> e = (2, 3)
>>> G.add_edge(*e)  # unpack edge tuple*
```

by adding a list of edges,

```
>>> G.add_edges_from([(1, 2), (1, 3)])
```

or by adding any *ebunch* of edges. An *ebunch* is any iterable container of edge-tuples. An edge-tuple can be a 2-tuple of nodes or a 3-tuple with 2 nodes followed by an edge attribute dictionary, e.g., `(2, 3, {'weight': 3.1415})`. Edge attributes are discussed further *below*.

```
>>> G.add_edges_from(H.edges)
```

There are no complaints when adding existing nodes or edges. For example, after removing all nodes and edges,

```
>>> G.clear()
```

we add new nodes/edges and NetworkX quietly ignores any that are already present.

```
>>> G.add_edges_from([(1, 2), (1, 3)])
>>> G.add_node(1)
>>> G.add_edge(1, 2)
>>> G.add_node("spam")        # adds node "spam"
>>> G.add_nodes_from("spam")  # adds 4 nodes: 's', 'p', 'a', 'm'
>>> G.add_edge(3, 'm')
```

At this stage the graph G consists of 8 nodes and 3 edges, as can be seen by:

```
>>> G.number_of_nodes()
8
>>> G.number_of_edges()
3
```

Note: The order of adjacency reporting (e.g., *G.adj*, *G.successors*, *G.predecessors*) is the order of edge addition. However, the order of G.edges is the order of the adjacencies which includes both the order of the nodes and each node's adjacencies. See example below:

```
>>> DG = nx.DiGraph()
>>> DG.add_edge(2, 1)   # adds the nodes in order 2, 1
>>> DG.add_edge(1, 3)
>>> DG.add_edge(2, 4)
>>> DG.add_edge(1, 2)
>>> assert list(DG.successors(2)) == [1, 4]
>>> assert list(DG.edges) == [(2, 1), (2, 4), (1, 3), (1, 2)]
```

A.4 Examining elements of a graph

We can examine the nodes and edges. Four basic graph properties facilitate reporting: `G.nodes`, `G.edges`, `G.adj` and `G.degree`. These are set-like views of the nodes, edges, neighbors (adjacencies), and degrees of nodes in a graph. They offer a continually updated read-only view into the graph structure. They are also dict-like in that you can look up node and edge data attributes via the views and iterate with data attributes using methods `.items()`, `.data()`. If you want a specific container type instead of a view, you can specify one. Here we use lists, though sets, dicts, tuples and other containers may be better in other contexts.

```
>>> list(G.nodes)
[1, 2, 3, 'spam', 's', 'p', 'a', 'm']
>>> list(G.edges)
[(1, 2), (1, 3), (3, 'm')]
>>> list(G.adj[1])  # or list(G.neighbors(1))
[2, 3]
>>> G.degree[1]  # the number of edges incident to 1
2
```

One can specify to report the edges and degree from a subset of all nodes using an *nbunch*. An *nbunch* is any of: `None` (meaning all nodes), a node, or an iterable container of nodes that is not itself a node in the graph.

```
>>> G.edges([2, 'm'])
EdgeDataView([(2, 1), ('m', 3)])
>>> G.degree([2, 3])
DegreeView({2: 1, 3: 2})
```

A.5 Removing elements from a graph

One can remove nodes and edges from the graph in a similar fashion to adding. Use methods *Graph.remove_node()*, *Graph.remove_nodes_from()*, *Graph.remove_edge()* and *Graph.remove_edges_from()*, e.g.

```
>>> G.remove_node(2)
>>> G.remove_nodes_from("spam")
>>> list(G.nodes)
[1, 3, 'spam']
>>> G.remove_edge(1, 3)
```

A.6 Using the graph constructors

Graph objects do not have to be built up incrementally - data specifying graph structure can be passed directly to the constructors of the various graph classes. When creating a graph structure by instantiating one of the graph classes you can specify data in several formats.

```
>>> G.add_edge(1, 2)
>>> H = nx.DiGraph(G)  # create a DiGraph using the connections from G
>>> list(H.edges())
[(1, 2), (2, 1)]
>>> edgelist = [(0, 1), (1, 2), (2, 3)]
>>> H = nx.Graph(edgelist)  # create a graph from an edge list
>>> list(H.edges())
[(0, 1), (1, 2), (2, 3)]
>>> adjacency_dict = {0: (1, 2), 1: (0, 2), 2: (0, 1)}
>>> H = nx.Graph(adjacency_dict)  # create a Graph dict mapping nodes to nbrs
>>> list(H.edges())
[(0, 1), (0, 2), (1, 2)]
```

A.7 What to use as nodes and edges

You might notice that nodes and edges are not specified as NetworkX objects. This leaves you free to use meaningful items as nodes and edges. The most common choices are numbers or strings, but a node can be any hashable object (except `None`), and an edge can be associated with any object `x` using `G.add_edge(n1, n2, object=x)`.

As an example, `n1` and `n2` could be protein objects from the RCSB Protein Data Bank, and `x` could refer to an XML record of publications detailing experimental observations of their interaction.

We have found this power quite useful, but its abuse can lead to surprising behavior unless one is familiar with Python. If in doubt, consider using *convert_node_labels_to_integers()* to obtain a more traditional graph with integer labels.

A.8 Accessing edges and neighbors

In addition to the views *Graph.edges*, and *Graph.adj*, access to edges and neighbors is possible using subscript notation.

```
>>> G = nx.Graph([(1, 2, {"color": "yellow"})])
>>> G[1]  # same as G.adj[1]
AtlasView({2: {'color': 'yellow'}})
>>> G[1][2]
{'color': 'yellow'}
>>> G.edges[1, 2]
{'color': 'yellow'}
```

You can get/set the attributes of an edge using subscript notation if the edge already exists.

```
>>> G.add_edge(1, 3)
>>> G[1][3]['color'] = "blue"
>>> G.edges[1, 2]['color'] = "red"
>>> G.edges[1, 2]
{'color': 'red'}
```

Fast examination of all (node, adjacency) pairs is achieved using `G.adjacency()`, or `G.adj.items()`. Note that for undirected graphs, adjacency iteration sees each edge twice.

```
>>> FG = nx.Graph()
>>> FG.add_weighted_edges_from([(1, 2, 0.125), (1, 3, 0.75), (2, 4, 1.2), (3, 4, 0.
↪375)])
>>> for n, nbrs in FG.adj.items():
...    for nbr, eattr in nbrs.items():
...        wt = eattr['weight']
...        if wt < 0.5: print(f"({n}, {nbr}, {wt:.3})")
(1, 2, 0.125)
(2, 1, 0.125)
(3, 4, 0.375)
(4, 3, 0.375)
```

Convenient access to all edges is achieved with the edges property.

```
>>> for (u, v, wt) in FG.edges.data('weight'):
...     if wt < 0.5:
...         print(f"({u}, {v}, {wt:.3})")
(1, 2, 0.125)
(3, 4, 0.375)
```

A.9 Adding attributes to graphs, nodes, and edges

Attributes such as weights, labels, colors, or whatever Python object you like, can be attached to graphs, nodes, or edges.

Each graph, node, and edge can hold key/value attribute pairs in an associated attribute dictionary (the keys must be hashable). By default these are empty, but attributes can be added or changed using `add_edge`, `add_node` or direct manipulation of the attribute dictionaries named `G.graph`, `G.nodes`, and `G.edges` for a graph `G`.

A.9.1 Graph attributes

Assign graph attributes when creating a new graph

```
>>> G = nx.Graph(day="Friday")
>>> G.graph
{'day': 'Friday'}
```

Or you can modify attributes later

```
>>> G.graph['day'] = "Monday"
>>> G.graph
{'day': 'Monday'}
```

A.9.2 Node attributes

Add node attributes using `add_node()`, `add_nodes_from()`, or `G.nodes`

```
>>> G.add_node(1, time='5pm')
>>> G.add_nodes_from([3], time='2pm')
>>> G.nodes[1]
{'time': '5pm'}
>>> G.nodes[1]['room'] = 714
>>> G.nodes.data()
NodeDataView({1: {'time': '5pm', 'room': 714}, 3: {'time': '2pm'}})
```

Note that adding a node to `G.nodes` does not add it to the graph, use `G.add_node()` to add new nodes. Similarly for edges.

A.9.3 Edge Attributes

Add/change edge attributes using `add_edge()`, `add_edges_from()`, or subscript notation.

```
>>> G.add_edge(1, 2, weight=4.7 )
>>> G.add_edges_from([(3, 4), (4, 5)], color='red')
>>> G.add_edges_from([(1, 2, {'color': 'blue'}), (2, 3, {'weight': 8})])
>>> G[1][2]['weight'] = 4.7
>>> G.edges[3, 4]['weight'] = 4.2
```

The special attribute `weight` should be numeric as it is used by algorithms requiring weighted edges.

A.10 Directed graphs

The *DiGraph* class provides additional methods and properties specific to directed edges, e.g., *DiGraph.out_edges*, *DiGraph.in_degree*, *DiGraph.predecessors*, *DiGraph.successors* etc. To allow algorithms to work with both classes easily, the directed versions of *neighbors* is equivalent to *successors* while *degree* reports the sum of *in_degree* and *out_degree* even though that may feel inconsistent at times.

```
>>> DG = nx.DiGraph()
>>> DG.add_weighted_edges_from([(1, 2, 0.5), (3, 1, 0.75)])
>>> DG.out_degree(1, weight='weight')
0.5
>>> DG.degree(1, weight='weight')
1.25
>>> list(DG.successors(1))
[2]
>>> list(DG.neighbors(1))
[2]
```

Some algorithms work only for directed graphs and others are not well defined for directed graphs. Indeed the tendency to lump directed and undirected graphs together is dangerous. If you want to treat a directed graph as undirected for some measurement you should probably convert it using *Graph.to_undirected()* or with

```
>>> H = nx.Graph(G)  # create an undirected graph H from a directed graph G
```

A.11 Multigraphs

NetworkX provides classes for graphs which allow multiple edges between any pair of nodes. The *MultiGraph* and *MultiDiGraph* classes allow you to add the same edge twice, possibly with different edge data. This can be powerful for some applications, but many algorithms are not well defined on such graphs. Where results are well defined, e.g., *MultiGraph.degree()* we provide the function. Otherwise you should convert to a standard graph in a way that makes the measurement well defined.

```
>>> MG = nx.MultiGraph()
>>> MG.add_weighted_edges_from([(1, 2, 0.5), (1, 2, 0.75), (2, 3, 0.5)])
>>> dict(MG.degree(weight='weight'))
{1: 1.25, 2: 1.75, 3: 0.5}
>>> GG = nx.Graph()
>>> for n, nbrs in MG.adjacency():
...    for nbr, edict in nbrs.items():
...        minvalue = min([d['weight'] for d in edict.values()])
...        GG.add_edge(n, nbr, weight = minvalue)
...
>>> nx.shortest_path(GG, 1, 3)
[1, 2, 3]
```

A.12 Graph generators and graph operations

In addition to constructing graphs node-by-node or edge-by-edge, they can also be generated by

A.12.1 1. Applying classic graph operations, such as:

subgraph(G, nbunch)	Returns the subgraph induced on nodes in nbunch.
union(G, H[, rename])	Combine graphs G and H.
disjoint_union(G, H)	Combine graphs G and H.
cartesian_product(G, H)	Returns the Cartesian product of G and H.
compose(G, H)	Compose graph G with H by combining nodes and edges into a single graph.
complement(G)	Returns the graph complement of G.
create_empty_copy(G[, with_data])	Returns a copy of the graph G with all of the edges removed.
to_undirected(graph)	Returns an undirected view of the graph `graph`.
to_directed(graph)	Returns a directed view of the graph `graph`.

A.12.2 2. Using a call to one of the classic small graphs, e.g.,

petersen_graph([create_using])	Returns the Petersen graph.
tutte_graph([create_using])	Returns the Tutte graph.
sedgewick_maze_graph([create_using])	Return a small maze with a cycle.
tetrahedral_graph([create_using])	Returns the 3-regular Platonic Tetrahedral graph.

A.12.3 3. Using a (constructive) generator for a classic graph, e.g.,

complete_graph(n[, create_using])	Return the complete graph `K_n` with n nodes.
complete_bipartite_graph(n1, n2[, create_using])	Returns the complete bipartite graph `K_{n_1,n_2}`.
barbell_graph(m1, m2[, create_using])	Returns the Barbell Graph: two complete graphs connected by a path.
lollipop_graph(m, n[, create_using])	Returns the Lollipop Graph; `K_m` connected to `P_n`.

like so:

```
>>> K_5 = nx.complete_graph(5)
>>> K_3_5 = nx.complete_bipartite_graph(3, 5)
>>> barbell = nx.barbell_graph(10, 10)
>>> lollipop = nx.lollipop_graph(10, 20)
```

A.12.4 4. Using a stochastic graph generator, e.g,

erdos_renyi_graph(n, p[, seed, directed])	Returns a $G_{n,p}$ random graph, also known as an Erdős-Rényi graph or a binomial graph.
watts_strogatz_graph(n, k, p[, seed])	Returns a Watts–Strogatz small-world graph.
barabasi_albert_graph(n, m[, seed, ...])	Returns a random graph using Barabási–Albert preferential attachment
random_lobster(n, p1, p2[, seed])	Returns a random lobster graph.

like so:

```
>>> er = nx.erdos_renyi_graph(100, 0.15)
>>> ws = nx.watts_strogatz_graph(30, 3, 0.1)
>>> ba = nx.barabasi_albert_graph(100, 5)
>>> red = nx.random_lobster(100, 0.9, 0.9)
```

A.12.5 5. Reading a graph stored in a file using common graph formats

NetworkX supports many popular formats, such as edge lists, adjacency lists, GML, GraphML, LEDA and others.

```
>>> nx.write_gml(red, "path.to.file")
>>> mygraph = nx.read_gml("path.to.file")
```

For details on graph formats see *Reading and writing graphs* and for graph generator functions see *Graph generators*

A.13 Analyzing graphs

The structure of `G` can be analyzed using various graph-theoretic functions such as:

```
>>> G = nx.Graph()
>>> G.add_edges_from([(1, 2), (1, 3)])
>>> G.add_node("spam")       # adds node "spam"
>>> list(nx.connected_components(G))
[{1, 2, 3}, {'spam'}]
>>> sorted(d for n, d in G.degree())
[0, 1, 1, 2]
>>> nx.clustering(G)
{1: 0, 2: 0, 3: 0, 'spam': 0}
```

Some functions with large output iterate over (node, value) 2-tuples. These are easily stored in a `dict` structure if you desire.

```
>>> sp = dict(nx.all_pairs_shortest_path(G))
>>> sp[3]
{3: [3], 1: [3, 1], 2: [3, 1, 2]}
```

See *Algorithms* for details on graph algorithms supported.

A.14 Drawing graphs

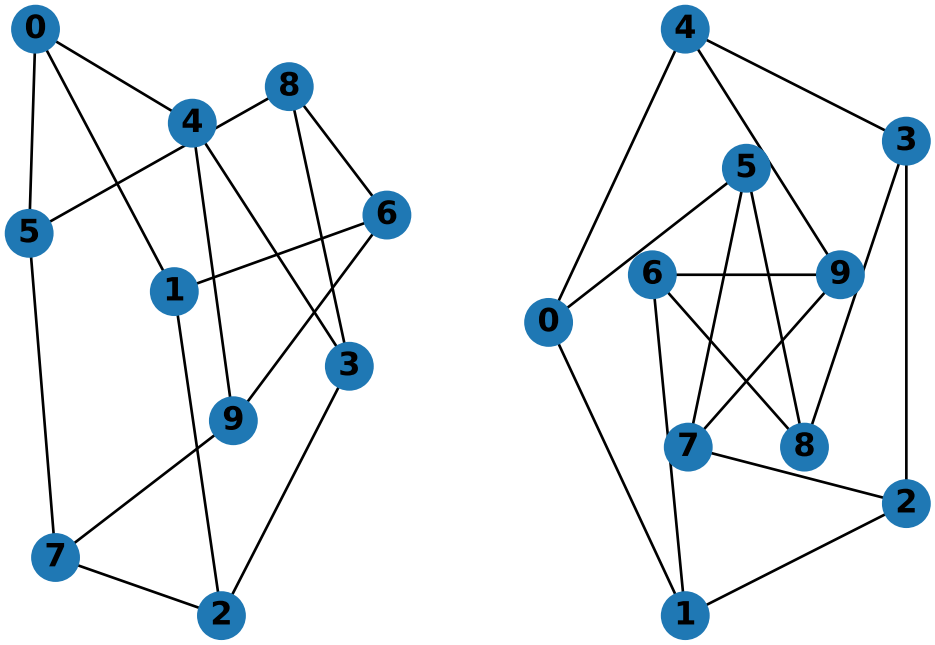
NetworkX is not primarily a graph drawing package but basic drawing with Matplotlib as well as an interface to use the open source Graphviz software package are included. These are part of the *networkx.drawing* module and will be imported if possible.

First import Matplotlib's plot interface (pylab works too)

```
>>> import matplotlib.pyplot as plt
```

To test if the import of *nx_pylab* was successful draw `G` using one of

```
>>> G = nx.petersen_graph()
>>> subax1 = plt.subplot(121)
>>> nx.draw(G, with_labels=True, font_weight='bold')
>>> subax2 = plt.subplot(122)
>>> nx.draw_shell(G, nlist=[range(5, 10), range(5)], with_labels=True, font_weight=
↪'bold')
```

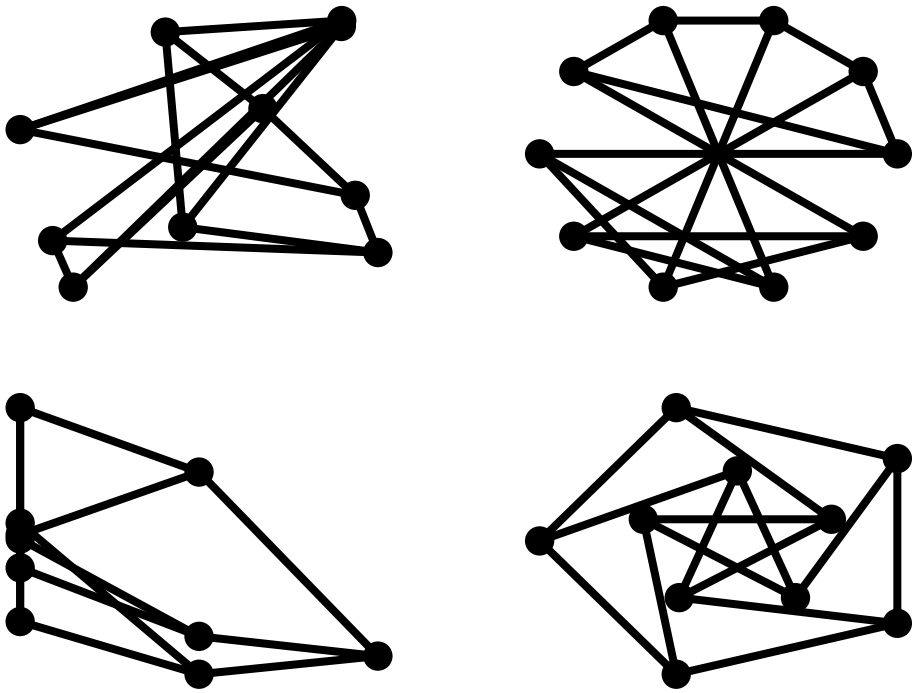


when drawing to an interactive display. Note that you may need to issue a Matplotlib

```
>>> plt.show()
```

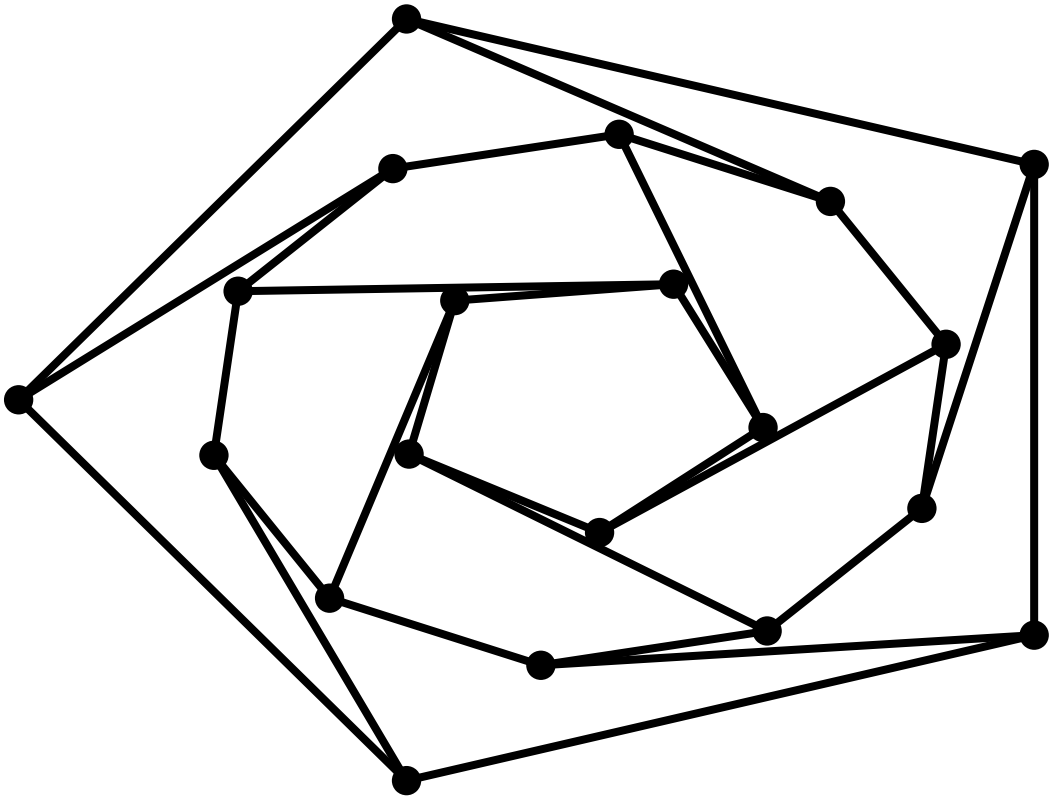
command if you are not using matplotlib in interactive mode.

```
>>> options = {
...     'node_color': 'black',
...     'node_size': 100,
...     'width': 3,
... }
>>> subax1 = plt.subplot(221)
>>> nx.draw_random(G, **options)
>>> subax2 = plt.subplot(222)
>>> nx.draw_circular(G, **options)
>>> subax3 = plt.subplot(223)
>>> nx.draw_spectral(G, **options)
>>> subax4 = plt.subplot(224)
>>> nx.draw_shell(G, nlist=[range(5,10), range(5)], **options)
```



You can find additional options via *draw_networkx()* and layouts via the *layout module*. You can use multiple shells with *draw_shell()*.

```
>>> G = nx.dodecahedral_graph()
>>> shells = [[2, 3, 4, 5, 6], [8, 1, 0, 19, 18, 17, 16, 15, 14, 7], [9, 10, 11, 12,␣
↪13]]
>>> nx.draw_shell(G, nlist=shells, **options)
```



To save drawings to a file, use, for example

```
>>> nx.draw(G)
>>> plt.savefig("path.png")
```

This function writes to the file `path.png` in the local directory. If Graphviz and PyGraphviz or pydot, are available on your system, you can also use *networkx.drawing.nx_agraph.graphviz_layout* or *networkx.drawing.nx_pydot.graphviz_layout* to get the node positions, or write the graph in dot format for further processing.

```
>>> from networkx.drawing.nx_pydot import write_dot
>>> pos = nx.nx_agraph.graphviz_layout(G)
>>> nx.draw(G, pos=pos)
>>> write_dot(G, 'file.dot')
```

See *Drawing* for additional details.

- ;
- ;
- .

BIBLIOGRAPHY

[1] White, Douglas R., and Mark Newman. 2001 A Fast Algorithm for Node-Independent Paths. Santa Fe Institute Working Paper #01-07-035 http://eclectic.ss.uci.edu/~drwhite/working.pdf

[1] White, Douglas R., and Mark Newman. 2001 A Fast Algorithm for Node-Independent Paths. Santa Fe Institute Working Paper #01-07-035 http://eclectic.ss.uci.edu/~drwhite/working.pdf

[1] White, Douglas R., and Mark Newman. 2001 A Fast Algorithm for Node-Independent Paths. Santa Fe Institute Working Paper #01-07-035 http://eclectic.ss.uci.edu/~drwhite/working.pdf

[1] Torrents, J. and F. Ferraro (2015) Structural Cohesion: Visualization and Heuristics for Fast Computation. https://arxiv.org/pdf/1503.04476v1

[2] White, Douglas R., and Mark Newman (2001) A Fast Algorithm for Node-Independent Paths. Santa Fe Institute Working Paper #01-07-035 https://www.santafe.edu/research/results/working-papers/fast-approximation-algorithms-for-finding-node-ind

[3] Moody, J. and D. White (2003). Social cohesion and embeddedness: A hierarchical conception of social groups. American Sociological Review 68(1), 103–28. https://doi.org/10.2307/3088904

[1] Wikipedia: Independent set

[2] Boppana, R., & Halldórsson, M. M. (1992). Approximating maximum independent sets by excluding subgraphs. BIT Numerical Mathematics, 32(2), 180–196. Springer.

[1] Boppana, R., & Halldórsson, M. M. (1992). Approximating maximum independent sets by excluding subgraphs. BIT Numerical Mathematics, 32(2), 180–196. Springer. doi:10.1007/BF01994876

[1] Boppana, R., & Halldórsson, M. M. (1992). Approximating maximum independent sets by excluding subgraphs. BIT Numerical Mathematics, 32(2), 180–196. Springer.

[1] Pattabiraman, Bharath, et al. "Fast Algorithms for the Maximum Clique Problem on Massive Graphs with Applications to Overlapping Community Detection." *Internet Mathematics* 11.4-5 (2015): 421–448. <https://doi.org/10.1080/15427951.2014.986778>

[1] Schank, Thomas, and Dorothea Wagner. Approximating clustering coefficient and transitivity. Universität Karlsruhe, Fakultät für Informatik, 2004. https://doi.org/10.5445/IR/1000001239

[1] Magnien, Clémence, Matthieu Latapy, and Michel Habib. *Fast computation of empirically tight bounds for the diameter of massive graphs.* Journal of Experimental Algorithmics (JEA), 2009. https://arxiv.org/pdf/0904.2728.pdf

[2] Crescenzi, Pierluigi, Roberto Grossi, Leonardo Lanzi, and Andrea Marino. *On computing the diameter of real-world directed (weighted) graphs.* International Symposium on Experimental Algorithms. Springer, Berlin, Heidelberg, 2012. https://courses.cs.ut.ee/MTAT.03.238/2014_fall/uploads/Main/diameter.pdf

[1] Vazirani, Vijay V. *Approximation Algorithms*. Springer Science & Business Media, 2001.

[1] Vazirani, Vijay Approximation Algorithms (2001)

[1] Steiner_tree_problem on Wikipedia. https://en.wikipedia.org/wiki/Steiner_tree_problem

[2] Kou, L., G. Markowsky, and L. Berman. 1981. 'A Fast Algorithm for Steiner Trees'. Acta Informatica 15 (2): 141–45. https://doi.org/10.1007/BF00288961.

[3] Mehlhorn, Kurt. 1988. 'A Faster Approximation Algorithm for the Steiner Problem in Graphs'. Information Processing Letters 27 (3): 125–28. https://doi.org/10.1016/0020-0190(88)90066-X.

[1] Christofides, Nicos. "Worst-case analysis of a new heuristic for the travelling salesman problem." No. RR-388. Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, 1976.

[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

[1] Hans L. Bodlaender and Arie M. C. A. Koster. 2010. "Treewidth computations I.Upper bounds". Inf. Comput. 208, 3 (March 2010),259-275. http://dx.doi.org/10.1016/j.ic.2009.03.008

[2] Hans L. Bodlaender. "Discovering Treewidth". Institute of Information and Computing Sciences, Utrecht University. Technical Report UU-CS-2005-018. http://www.cs.uu.nl

[3] K. Wang, Z. Lu, and J. Hicks *Treewidth*. https://web.archive.org/web/20210507025929/http://web.eecs.utk.edu/~cphill25/cs594_spring2015_projects/treewidth.pdf

[1] Bar-Yehuda, R., and Even, S. (1985). "A local-ratio theorem for approximating the weighted vertex cover problem." *Annals of Discrete Mathematics*, 25, 27–46 <http://www.cs.technion.ac.il/~reuven/PDF/vc_lr.pdf>

[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).

[1] M. E. J. Newman, Mixing patterns in networks, Physical Review E, 67 026126, 2003

[1] M. E. J. Newman, Mixing patterns in networks Physical Review E, 67 026126, 2003

[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).

[1] A. Barrat, M. Barthélemy, R. Pastor-Satorras, and A. Vespignani, "The architecture of complex weighted networks". PNAS 101 (11): 3747–3752 (2004).

[1] A. Barrat, M. Barthélemy, R. Pastor-Satorras, and A. Vespignani, "The architecture of complex weighted networks". PNAS 101 (11): 3747–3752 (2004).

[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

[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>.

[1] Richard Manning Karp: An algorithm to Solve the m x n Assignment Problem in Expected Time O(mn log n). Networks, 10(2):143–152, 1980.

[1] https://en.wikipedia.org/wiki/Adjacency_matrix#Adjacency_matrix_of_a_bipartite_graph

[2] Scipy Dev. References, "Sparse Matrices", https://docs.scipy.org/doc/scipy/reference/sparse.html

[1] Borgatti, S.P. and Halgin, D. In press. "Analyzing Affiliation Networks". In Carrington, P. and Scott, J. (eds) The Sage Handbook of Social Network Analysis. Sage Publications.

[1] Scientific collaboration networks: II. Shortest paths, weighted networks, and centrality, M. E. J. Newman, Phys. Rev. E 64, 016132 (2001).

[1] Borgatti, S.P. and Halgin, D. In press. Analyzing Affiliation Networks. In Carrington, P. and Scott, J. (eds) The Sage Handbook of Social Network Analysis. Sage Publications.

[1] E. Estrada and J. A. Rodríguez-Velázquez, "Spectral measures of bipartivity in complex networks", PhysRev E 72, 046105 (2005)

[1] Latapy, Matthieu, Clémence Magnien, and Nathalie Del Vecchio (2008). Basic notions for the analysis of large two-mode networks. Social Networks 30(1), 31–48.

[1] Latapy, Matthieu, Clémence Magnien, and Nathalie Del Vecchio (2008). Basic notions for the analysis of large two-mode networks. Social Networks 30(1), 31–48.

[1] Latapy, Matthieu, Clémence Magnien, and Nathalie Del Vecchio (2008). Basic notions for the analysis of large two-mode networks. Social Networks 30(1), 31–48.

[1] Robins, G. and M. Alexander (2004). Small worlds among interlocking directors: Network structure and distance in bipartite graphs. Computational & Mathematical Organization Theory 10(1), 69–94.

[1] Latapy, Matthieu, Clémence Magnien, and Nathalie Del Vecchio (2008). Basic notions for the analysis of large two-mode networks. Social Networks 30(1), 31–48.

[1] Borgatti, S.P. and Halgin, D. In press. "Analyzing Affiliation Networks". In Carrington, P. and Scott, J. (eds) The Sage Handbook of Social Network Analysis. Sage Publications. https://dx.doi.org/10.4135/9781446294413.n28

[1] Borgatti, S.P. and Halgin, D. In press. "Analyzing Affiliation Networks". In Carrington, P. and Scott, J. (eds) The Sage Handbook of Social Network Analysis. Sage Publications. https://dx.doi.org/10.4135/9781446294413.n28

[1] Borgatti, S.P. and Halgin, D. In press. "Analyzing Affiliation Networks". In Carrington, P. and Scott, J. (eds) The Sage Handbook of Social Network Analysis. Sage Publications. https://dx.doi.org/10.4135/9781446294413.n28

[1] Guillaume, J.L. and Latapy, M., Bipartite graphs as models of complex networks. Physica A: Statistical Mechanics and its Applications, 2006, 371(2), pp.795-813.

[2] Jean-Loup Guillaume and Matthieu Latapy, Bipartite structure of all complex networks, Inf. Process. Lett. 90, 2004, pg. 215-221 https://doi.org/10.1016/j.ipl.2004.03.007

[1] Vladimir Batagelj and Ulrik Brandes, "Efficient generation of large random networks", Phys. Rev. E, 71, 036113, 2005.

[1] https://en.wikipedia.org/wiki/Bridge_%28graph_theory%29#Bridge-Finding_with_Chain_Decompositions

[1] Phillip Bonacich. "Power and Centrality: A Family of Measures." *American Journal of Sociology* 92(5):1170–1182, 1986 <http://www.leonidzhukov.net/hse/2014/socialnetworks/papers/Bonacich-Centrality.pdf>

[2] Mark E. J. Newman. *Networks: An Introduction.* Oxford University Press, USA, 2010, pp. 169.

[1] Phillip Bonacich: Power and Centrality: A Family of Measures. American Journal of Sociology 92(5):1170–1182, 1986 http://www.leonidzhukov.net/hse/2014/socialnetworks/papers/Bonacich-Centrality.pdf

[2] Mark E. J. Newman: Networks: An Introduction. Oxford University Press, USA, 2010, pp. 169.

[1] Mark E. J. Newman: Networks: An Introduction. Oxford University Press, USA, 2010, p. 720.

[2] Leo Katz: A New Status Index Derived from Sociometric Index. Psychometrika 18(1):39–43, 1953 https://link.springer.com/content/pdf/10.1007/BF02289026.pdf

[1] Mark E. J. Newman: Networks: An Introduction. Oxford University Press, USA, 2010, p. 173.

[2] Leo Katz: A New Status Index Derived from Sociometric Index. Psychometrika 18(1):39–43, 1953 https://link.springer.com/content/pdf/10.1007/BF02289026.pdf

[1] Linton C. Freeman: Centrality in networks: I. Conceptual clarification. Social Networks 1:215-239, 1979. https://doi.org/10.1016/0378-8733(78)90021-7

[2] pg. 201 of Wasserman, S. and Faust, K., Social Network Analysis: Methods and Applications, 1994, Cambridge University Press.

[1] Freeman, L.C., 1979. Centrality in networks: I. Conceptual clarification. Social Networks 1, 215–239. https://doi.org/10.1016/0378-8733(78)90021-7

[2] Sariyuce, A.E. ; Kaya, K. ; Saule, E. ; Catalyiirek, U.V. Incremental Algorithms for Closeness Centrality. 2013 IEEE International Conference on Big Data http://sariyuce.com/papers/bigdata13.pdf

[1] Ulrik Brandes and Daniel Fleischer, Centrality Measures Based on Current Flow. Proc. 22nd Symp. Theoretical Aspects of Computer Science (STACS '05). LNCS 3404, pp. 533-544. Springer-Verlag, 2005. https://doi.org/10.1007/978-3-540-31856-9_44

[2] Karen Stephenson and Marvin Zelen: Rethinking centrality: Methods and examples. Social Networks 11(1):1-37, 1989. https://doi.org/10.1016/0378-8733(89)90016-6

[1] Ulrik Brandes and Daniel Fleischer, Centrality Measures Based on Current Flow. Proc. 22nd Symp. Theoretical Aspects of Computer Science (STACS '05). LNCS 3404, pp. 533-544. Springer-Verlag, 2005. https://doi.org/10.1007/978-3-540-31856-9_44

[2] Karen Stephenson and Marvin Zelen: Rethinking centrality: Methods and examples. Social Networks 11(1):1-37, 1989. https://doi.org/10.1016/0378-8733(89)90016-6

[1] 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

[2] Ulrik Brandes: On Variants of Shortest-Path Betweenness Centrality and their Generic Computation. Social Networks 30(2):136-145, 2008. https://doi.org/10.1016/j.socnet.2007.11.001

[3] Ulrik Brandes and Christian Pich: Centrality Estimation in Large Networks. International Journal of Bifurcation and Chaos 17(7):2303-2318, 2007. https://dx.doi.org/10.1142/S0218127407018403

[4] Linton C. Freeman: A set of measures of centrality based on betweenness. Sociometry 40: 35–41, 1977 https://doi.org/10.2307/3033543

[1] 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

[2] Ulrik Brandes: On Variants of Shortest-Path Betweenness Centrality and their Generic Computation. Social Networks 30(2):136-145, 2008. https://doi.org/10.1016/j.socnet.2007.11.001

[1] A Faster Algorithm for Betweenness Centrality. Ulrik Brandes, Journal of Mathematical Sociology 25(2):163-177, 2001. https://doi.org/10.1080/0022250X.2001.9990249

[2] Ulrik Brandes: On Variants of Shortest-Path Betweenness Centrality and their Generic Computation. Social Networks 30(2):136-145, 2008. https://doi.org/10.1016/j.socnet.2007.11.001

[1] 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

[2] Ulrik Brandes: On Variants of Shortest-Path Betweenness Centrality and their Generic Computation. Social Networks 30(2):136-145, 2008. https://doi.org/10.1016/j.socnet.2007.11.001

[1] Centrality Measures Based on Current Flow. Ulrik Brandes and Daniel Fleischer, Proc. 22nd Symp. Theoretical Aspects of Computer Science (STACS '05). LNCS 3404, pp. 533-544. Springer-Verlag, 2005. https://doi.org/10.1007/978-3-540-31856-9_44

[2] A measure of betweenness centrality based on random walks, M. E. J. Newman, Social Networks 27, 39-54 (2005).

[1] Centrality Measures Based on Current Flow. Ulrik Brandes and Daniel Fleischer, Proc. 22nd Symp. Theoretical Aspects of Computer Science (STACS '05). LNCS 3404, pp. 533-544. Springer-Verlag, 2005. https://doi.org/10.1007/978-3-540-31856-9_44

[2] A measure of betweenness centrality based on random walks, M. E. J. Newman, Social Networks 27, 39-54 (2005).

[1] Ulrik Brandes and Daniel Fleischer: Centrality Measures Based on Current Flow. Proc. 22nd Symp. Theoretical Aspects of Computer Science (STACS '05). LNCS 3404, pp. 533-544. Springer-Verlag, 2005. https://doi.org/10.1007/978-3-540-31856-9_44

[1] Centrality Measures Based on Current Flow. Ulrik Brandes and Daniel Fleischer, Proc. 22nd Symp. Theoretical Aspects of Computer Science (STACS '05). LNCS 3404, pp. 533-544. Springer-Verlag, 2005. https://doi.org/10.1007/978-3-540-31856-9_44

[2] A measure of betweenness centrality based on random walks, M. E. J. Newman, Social Networks 27, 39-54 (2005).

[1] Centrality Measures Based on Current Flow. Ulrik Brandes and Daniel Fleischer, Proc. 22nd Symp. Theoretical Aspects of Computer Science (STACS '05). LNCS 3404, pp. 533-544. Springer-Verlag, 2005. https://doi.org/10.1007/978-3-540-31856-9_44

[2] A measure of betweenness centrality based on random walks, M. E. J. Newman, Social Networks 27, 39-54 (2005).

[1] Ernesto Estrada, Desmond J. Higham, Naomichi Hatano, "Communicability Betweenness in Complex Networks" Physica A 388 (2009) 764-774. https://arxiv.org/abs/0905.4102

[1] M G Everett and S P Borgatti: The Centrality of Groups and Classes. Journal of Mathematical Sociology. 23(3): 181-201. 1999. http://www.analytictech.com/borgatti/group_centrality.htm

[2] Ulrik Brandes: On Variants of Shortest-Path Betweenness Centrality and their Generic Computation. Social Networks 30(2):136-145, 2008. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.72.9610&rep=rep1&type=pdf

[3] Sourav Medya et. al.: Group Centrality Maximization via Network Design. SIAM International Conference on Data Mining, SDM 2018, 126–134. https://sites.cs.ucsb.edu/~arlei/pubs/sdm18.pdf

[4] Rami Puzis, Yuval Elovici, and Shlomi Dolev. "Fast algorithm for successive computation of group betweenness centrality." https://journals.aps.org/pre/pdf/10.1103/PhysRevE.76.056709

[1] M G Everett and S P Borgatti: The Centrality of Groups and Classes. Journal of Mathematical Sociology. 23(3): 181-201. 1999. http://www.analytictech.com/borgatti/group_centrality.htm

[2] J. Zhao et. al.: Measuring and Maximizing Group Closeness Centrality over Disk Resident Graphs. WWW-Conference Proceedings, 2014. 689-694. https://doi.org/10.1145/2567948.2579356

[1] M G Everett and S P Borgatti: The Centrality of Groups and Classes. Journal of Mathematical Sociology. 23(3): 181-201. 1999. http://www.analytictech.com/borgatti/group_centrality.htm

[1] M G Everett and S P Borgatti: The Centrality of Groups and Classes. Journal of Mathematical Sociology. 23(3): 181-201. 1999. http://www.analytictech.com/borgatti/group_centrality.htm

[2] Rami Puzis, Yuval Elovici, and Shlomi Dolev: "Finding the Most Prominent Group in Complex Networks" AI communications 20(4): 287-296, 2007. https://www.researchgate.net/profile/Rami_Puzis2/publication/220308855

[3] Sourav Medya et. al.: Group Centrality Maximization via Network Design. SIAM International Conference on Data Mining, SDM 2018, 126–134. https://sites.cs.ucsb.edu/~arlei/pubs/sdm18.pdf

[4] Rami Puzis, Yuval Elovici, and Shlomi Dolev. “Fast algorithm for successive computation of group betweenness centrality.” https://journals.aps.org/pre/pdf/10.1103/PhysRevE.76.056709

[1] Mark E. J. Newman: Scientific collaboration networks. II. Shortest paths, weighted networks, and centrality. Physical Review E 64, 016132, 2001. http://journals.aps.org/pre/abstract/10.1103/PhysRevE.64.016132

[2] Kwang-Il Goh, Byungnam Kahng and Doochul Kim Universal behavior of Load Distribution in Scale-Free Networks. Physical Review Letters 87(27):1–4, 2001. https://doi.org/10.1103/PhysRevLett.87.278701

[1] Ernesto Estrada, Juan A. Rodriguez-Velazquez, “Subgraph centrality in complex networks”, Physical Review E 71, 056103 (2005). https://arxiv.org/abs/cond-mat/0504730

[1] Ernesto Estrada, Juan A. Rodriguez-Velazquez, “Subgraph centrality in complex networks”, Physical Review E 71, 056103 (2005). https://arxiv.org/abs/cond-mat/0504730

[1] E. Estrada, “Characterization of 3D molecular structure”, Chem. Phys. Lett. 319, 713 (2000). https://doi.org/10.1016/S0009-2614(00)00158-5

[2] José Antonio de la Peñaa, Ivan Gutman, Juan Rada, “Estimating the Estrada index”, Linear Algebra and its Applications. 427, 1 (2007). https://doi.org/10.1016/j.laa.2007.06.020

[1] Boldi, Paolo, and Sebastiano Vigna. “Axioms for centrality.” Internet Mathematics 10.3-4 (2014): 222-262.

[1] Romantic Partnerships and the Dispersion of Social Ties: A Network Analysis of Relationship Status on Facebook. Lars Backstrom, Jon Kleinberg. https://arxiv.org/pdf/1310.6753v1.pdf

[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

[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

[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

[1] Anne-Marie Kermarrec, Erwan Le Merrer, Bruno Sericola, Gilles Trédan “Second order centrality: Distributed assessment of nodes criticity in complex networks”, Elsevier Computer Communications 34(5):619-628, 2011.

[1] Stephen Levine (1980) J. theor. Biol. 83, 195-207

[1] Samuel Johnson, Virginia Dominguez-Garcia, Luca Donetti, Miguel A. Munoz (2014) PNAS “Trophic coherence determines food-web stability”

[1] Samuel Johnson, Virginia Dominguez-Garcia, Luca Donetti, Miguel A. Munoz (2014) PNAS “Trophic coherence determines food-web stability”

[1] Zhang, J.-X. et al. (2016). Identifying a set of influential spreaders in complex networks. Sci. Rep. 6, 27823; doi: 10.1038/srep27823.

[1] Jens M. Schmidt (2013). “A simple test on 2-vertex- and 2-edge-connectivity.” *Information Processing Letters*, 113, 241–244. Elsevier. <https://doi.org/10.1016/j.ipl.2013.01.016>

[1] R. E. Tarjan and M. Yannakakis, Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs, SIAM J. Comput., 13 (1984), pp. 566–579.

[1] https://en.wikipedia.org/wiki/Tree_decomposition#Treewidth

[1] Berry, Anne & Blair, Jean & Heggernes, Pinar & Peyton, Barry. (2004) Maximum Cardinality Search for Computing Minimal Triangulations of Graphs. Algorithmica. 39. 287-298. 10.1007/s00453-004-1084-3.

[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

[1] clique problem:: https://en.wikipedia.org/wiki/Clique_problem

[1] Yun Zhang, Abu-Khzam, F.N., Baldwin, N.E., Chesler, E.J., Langston, M.A., Samatova, N.F., "Genome-Scale Computational Approaches to Memory-Intensive Applications in Systems Biology". *Supercomputing*, 2005. Proceedings of the ACM/IEEE SC 2005 Conference, pp. 12, 12–18 Nov. 2005. <https://doi.org/10.1109/SC.2005.29>.

[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>

[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>

[1] Tavares, W.A., Neto, M.B.C., Rodrigues, C.D., Michelon, P.: Um algoritmo de branch and bound para o problema da clique máxima ponderada. Proceedings of XLVII SBPO 1 (2015).

[2] Warrent, Jeffrey S, Hicks, Illya V.: Combinatorial Branch-and-Bound for the Maximum Weight Independent Set Problem. Technical Report, Texas A&M University (2016).

[1] Generalizations of the clustering coefficient to weighted complex networks by J. Saramäki, M. Kivelä, J.-P. Onnela, K. Kaski, and J. Kertész, Physical Review E, 75 027105 (2007). http://jponnela.com/web_documents/a9.pdf

[2] Intensity and coherence of motifs in weighted complex networks by J. P. Onnela, J. Saramäki, J. Kertész, and K. Kaski, Physical Review E, 71(6), 065103 (2005).

[3] Generalization of Clustering Coefficients to Signed Correlation Networks by G. Costantini and M. Perugini, PloS one, 9(2), e88669 (2014).

[4] Clustering in complex directed networks by G. Fagiolo, Physical Review E, 76(2), 026107 (2007).

[1] Generalizations of the clustering coefficient to weighted complex networks by J. Saramäki, M. Kivelä, J.-P. Onnela, K. Kaski, and J. Kertész, Physical Review E, 75 027105 (2007). http://jponnela.com/web_documents/a9.pdf

[2] Marcus Kaiser, Mean clustering coefficients: the role of isolated nodes and leafs on clustering measures for small-world networks. https://arxiv.org/abs/0802.2512

[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. https://arxiv.org/abs/0710.0117v1

[1] Networks with arbitrary edge multiplicities by V. Zlatić, D. Garlaschelli and G. Caldarelli, EPL (Europhysics Letters), Volume 97, Number 2 (2012). https://iopscience.iop.org/article/10.1209/0295-5075/97/28005

[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.

[1] Kierstead, H. A., Kostochka, A. V., Mydlarz, M., & Szemerédi, E. (2010). A fast algorithm for equitable coloring. Combinatorica, 30(2), 217-224.

[1] Ernesto Estrada, Naomichi Hatano, "Communicability in complex networks", Phys. Rev. E 77, 036111 (2008). https://arxiv.org/abs/0707.0756

[1] Ernesto Estrada, Naomichi Hatano, "Communicability in complex networks", Phys. Rev. E 77, 036111 (2008). https://arxiv.org/abs/0707.0756

[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.

[1] Gergely Palla, Imre Derényi, Illés Farkas1, and Tamás Vicsek, Uncovering the overlapping community structure of complex networks in nature and society Nature 435, 814-818, 2005, doi:10.1038/nature03607

[1] Newman, M. E. J. "Networks: An Introduction", page 224 Oxford University Press 2011.

[2] Clauset, A., Newman, M. E., & Moore, C. "Finding community structure in very large networks." Physical Review E 70(6), 2004.

[3] Reichardt and Bornholdt "Statistical Mechanics of Community Detection" Phys. Rev. E74, 2006.

[4] Newman, M. E. J."Analysis of weighted networks" Physical Review E 70(5 Pt 2):056131, 2004.

[1] Raghavan, Usha Nandini, Réka Albert, and Soundar Kumara. "Near linear time algorithm to detect community structures in large-scale networks." Physical Review E 76.3 (2007): 036106.

[1] Cordasco, G., & Gargano, L. (2010, December). Community detection via semi-synchronous label propagation algorithms. In Business Applications of Social Network Analysis (BASNA), 2010 IEEE International Workshop on (pp. 1-8). IEEE.

[1] Blondel, V.D. et al. Fast unfolding of communities in large networks. J. Stat. Mech 10008, 1-12(2008). https://doi.org/10.1088/1742-5468/2008/10/P10008

[2] Traag, V.A., Waltman, L. & van Eck, N.J. From Louvain to Leiden: guaranteeing well-connected communities. Sci Rep 9, 5233 (2019). https://doi.org/10.1038/s41598-019-41695-z

[3] Nicolas Dugué, Anthony Perez. Directed Louvain : maximizing modularity in directed networks. [Research Report] Université d'Orléans. 2015. hal-01231784. https://hal.archives-ouvertes.fr/hal-01231784

[1] Blondel, V.D. et al. Fast unfolding of communities in large networks. J. Stat. Mech 10008, 1-12(2008)

[1] Parés F., Garcia-Gasulla D. et al. "Fluid Communities: A Competitive and Highly Scalable Community Detection Algorithm". [https://arxiv.org/pdf/1703.09307.pdf].

[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

[4] M. E. J. Newman, "Equivalence between modularity optimization and maximum likelihood methods for community detection" Phys. Rev. E 94, 052315, 2016. https://doi.org/10.1103/PhysRevE.94.052315

[1] Santo Fortunato. "Community Detection in Graphs". *Physical Reports*, Volume 486, Issue 3–5 pp. 75–174 <https://arxiv.org/abs/0906.0612>

[1] Depth-first search and linear graph algorithms, R. Tarjan SIAM Journal of Computing 1(2):146-160, (1972).

[2] On finding the strongly connected components in a directed graph. E. Nuutila and E. Soisalon-Soinen Information Processing Letters 49(1): 9-14, (1994)..

[1] Depth-first search and linear graph algorithms, R. Tarjan SIAM Journal of Computing 1(2):146-160, (1972).

[2] On finding the strongly connected components in a directed graph. E. Nuutila and E. Soisalon-Soinen Information Processing Letters 49(1): 9-14, (1994)..

[1] Hopcroft, J.; Tarjan, R. (1973). "Efficient algorithms for graph manipulation". Communications of the ACM 16: 372–378. doi:10.1145/362248.362272

[1] Hopcroft, J.; Tarjan, R. (1973). "Efficient algorithms for graph manipulation". Communications of the ACM 16: 372–378. doi:10.1145/362248.362272

[1] Hopcroft, J.; Tarjan, R. (1973). "Efficient algorithms for graph manipulation". Communications of the ACM 16: 372–378. doi:10.1145/362248.362272

[1] Hopcroft, J.; Tarjan, R. (1973). "Efficient algorithms for graph manipulation". Communications of the ACM 16: 372–378. doi:10.1145/362248.362272

[1] https://en.wikipedia.org/wiki/Bridge_%28graph_theory%29

[2] Wang, Tianhao, et al. (2015) A simple algorithm for finding all k-edge-connected components. http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0136264

[1] Zhou, Liu, et al. (2012) Finding maximal k-edge-connected subgraphs from a large graph. ACM International Conference on Extending Database Technology 2012 480-–491. https://openproceedings.org/2012/conf/edbt/ZhouLYLCL12.pdf

[1] Wang, Tianhao, et al. (2015) A simple algorithm for finding all k-edge-connected components. http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0136264

[1] Moody, J. and D. White (2003). Social cohesion and embeddedness: A hierarchical conception of social groups. American Sociological Review 68(1), 103–28. http://www2.asanet.org/journals/ASRFeb03MoodyWhite.pdf

[2] 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

[3] Torrents, J. and F. Ferraro (2015). Structural Cohesion: Visualization and Heuristics for Fast Computation. https://arxiv.org/pdf/1503.04476v1

[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

[1] Beineke, L., O. Oellermann, and R. Pippert (2002). The average connectivity of a graph. Discrete mathematics 252(1-3), 31-45. http://www.sciencedirect.com/science/article/pii/S0012365X01001807

[1] Abdol-Hossein Esfahanian. Connectivity Algorithms. http://www.cse.msu.edu/~cse835/Papers/Graph_connectivity_revised.pdf

[1] Abdol-Hossein Esfahanian. Connectivity Algorithms. http://www.cse.msu.edu/~cse835/Papers/Graph_connectivity_revised.pdf

[1] Kammer, Frank and Hanjo Taubig. Graph Connectivity. in Brandes and Erlebach, 'Network Analysis: Methodological Foundations', Lecture Notes in Computer Science, Volume 3418, Springer-Verlag, 2005. http://www.informatik.uni-augsburg.de/thi/personen/kammer/Graph_Connectivity.pdf

[1] Abdol-Hossein Esfahanian. Connectivity Algorithms. http://www.cse.msu.edu/~cse835/Papers/Graph_connectivity_revised.pdf

[1] Abdol-Hossein Esfahanian. Connectivity Algorithms. http://www.cse.msu.edu/~cse835/Papers/Graph_connectivity_revised.pdf

[1] Abdol-Hossein Esfahanian. Connectivity Algorithms. http://www.cse.msu.edu/~cse835/Papers/Graph_connectivity_revised.pdf

[1] Abdol-Hossein Esfahanian. Connectivity Algorithms. http://www.cse.msu.edu/~cse835/Papers/Graph_connectivity_revised.pdf

[1] Abdol-Hossein Esfahanian. Connectivity Algorithms. (this is a chapter, look for the reference of the book). http://www.cse.msu.edu/~cse835/Papers/Graph_connectivity_revised.pdf

[1] Kammer, Frank and Hanjo Taubig. Graph Connectivity. in Brandes and Erlebach, 'Network Analysis: Methodological Foundations', Lecture Notes in Computer Science, Volume 3418, Springer-Verlag, 2005. https://doi.org/10.1007/978-3-540-31955-9_7

[1] An O(m) Algorithm for Cores Decomposition of Networks Vladimir Batagelj and Matjaz Zaversnik, 2003. https://arxiv.org/abs/cs.DS/0310049

[1] An O(m) Algorithm for Cores Decomposition of Networks Vladimir Batagelj and Matjaz Zaversnik, 2003. https://arxiv.org/abs/cs.DS/0310049

[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

[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

[1] k -core (bootstrap) percolation on complex networks: Critical phenomena and nonlocal effects, A. V. Goltsev, S. N. Dorogovtsev, and J. F. F. Mendes, Phys. Rev. E 73, 056101 (2006) http://link.aps.org/doi/10.1103/PhysRevE.73.056101

[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.

[1] Multi-scale structure and topological anomaly detection via a new network statistic: The onion decomposition L. Hébert-Dufresne, J. A. Grochow, and A. Allard Scientific Reports 6, 31708 (2016) http://doi.org/10.1038/srep31708

[2] Percolation and the effective structure of complex networks A. Allard and L. Hébert-Dufresne Physical Review X 9, 011023 (2019) http://doi.org/10.1103/PhysRevX.9.011023

[1] Paton, K. An algorithm for finding a fundamental set of cycles of a graph. Comm. ACM 12, 9 (Sept 1969), 514-518.

[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] Enumerating the cycles of a digraph: a new preprocessing strategy. G. Loizou and P. Thanish, Information Sciences, v. 27, 163-182, 1982.

[3] 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.

[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

[1] Vadhan, Salil P. "Pseudorandomness." *Foundations and Trends in Theoretical Computer Science* 7.1–3 (2011): 1–336. <https://doi.org/10.1561/0400000010>

[1] David Gleich. *Hierarchical Directed Spectral Graph Partitioning*. <https://www.cs.purdue.edu/homes/dgleich/publications/Gleich%202005%20-%20hierarchical%20directed%20spectral.pdf>

[1] Fan Chung. *Spectral Graph Theory*. (CBMS Regional Conference Series in Mathematics, No. 92), American Mathematical Society, 1997, ISBN 0-8218-0315-8 <http://www.math.ucsd.edu/~fan/research/revised.html>

[1] Vadhan, Salil P. "Pseudorandomness." *Foundations and Trends in Theoretical Computer Science* 7.1–3 (2011): 1–336. <https://doi.org/10.1561/0400000010>

[1] Vadhan, Salil P. "Pseudorandomness." *Foundations and Trends in Theoretical Computer Science* 7.1–3 (2011): 1–336. <https://doi.org/10.1561/0400000010>

[1] David Gleich. *Hierarchical Directed Spectral Graph Partitioning*. <https://www.cs.purdue.edu/homes/dgleich/publications/Gleich%202005%20-%20hierarchical%20directed%20spectral.pdf>

[1] David Gleich. *Hierarchical Directed Spectral Graph Partitioning*. <https://www.cs.purdue.edu/homes/dgleich/publications/Gleich%202005%20-%20hierarchical%20directed%20spectral.pdf>

[1] Pearl, J. (2009). Causality. Cambridge: Cambridge University Press.

[2] Darwiche, A. (2009). Modeling and reasoning with Bayesian networks. Cambridge: Cambridge University Press.

[3] Shachter, R. D. (1998). Bayes-ball: 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 (pp. 480–487). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

[4] Koller, D., & Friedman, N. (2009). Probabilistic graphical models: principles and techniques. The MIT Press.

[5] https://en.wikipedia.org/wiki/Causal_Markov_condition

[1] Manber, U. (1989). *Introduction to Algorithms - A Creative Approach.* Addison-Wesley.

[1] Knuth, Donald E., Szwarcfiter, Jayme L. (1974). "A Structured Program to Generate All Topological Sorting Arrangements" Information Processing Letters, Volume 2, Issue 6, 1974, Pages 153-157, ISSN 0020-0190, https://doi.org/10.1016/0020-0190(74)90001-5. Elsevier (North-Holland), Amsterdam

[1] Manber, U. (1989). *Introduction to Algorithms - A Creative Approach.* Addison-Wesley.

[1] Jarvis, J. P.; Shier, D. R. (1996), "Graph-theoretic analysis of finite Markov chains," in Shier, D. R.; Wallenius, K. T., Applied Mathematical Modeling: A Multidisciplinary Approach, CRC Press.

[1] https://www.ics.uci.edu/~eppstein/PADS/PartialOrder.py

[1] Free Lattices, by R. Freese, J. Jezek and J. B. Nation, AMS, Vol 42, 1995, p. 226.

[1] Wikipedia "Resistance distance." https://en.wikipedia.org/wiki/Resistance_distance

[2] E. W. Weisstein "Resistance Distance." MathWorld–A Wolfram Web Resource https://mathworld.wolfram.com/ResistanceDistance.html

[3] V. S. S. Vos, "Methods for determining the effective resistance." Mestrado, Mathematisch Instituut Universiteit Leiden, 2016 https://www.universiteitleiden.nl/binaries/content/assets/science/mi/scripties/master/vos_vaya_master.pdf

[1] Brouwer, A. E.; Cohen, A. M.; and Neumaier, A. Distance-Regular Graphs. New York: Springer-Verlag, 1989.

[2] Weisstein, Eric W. "Distance-Regular Graph." http://mathworld.wolfram.com/Distance-RegularGraph.html

[1] Weisstein, Eric W. "Intersection Array." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/IntersectionArray.html

[1] Weisstein, Eric W. "Global Parameters." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/GlobalParameters.html

[1] K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm. Software Practice & Experience, 4:110, 2001.

[1] K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm. Software Practice & Experience, 4:110, 2001.

[1] https://en.wikipedia.org/wiki/Dominating_set

[2] Abdol-Hossein Esfahanian. Connectivity Algorithms. http://www.cse.msu.edu/~cse835/Papers/Graph_connectivity_revised.pdf

[1] https://en.wikipedia.org/wiki/Dominating_set

[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>

[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>

[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>

[1] J. Edmonds, E. L. Johnson. Matching, Euler tours and the Chinese postman. Mathematical programming, Volume 5, Issue 1 (1973), 111-114.

[2] https://en.wikipedia.org/wiki/Eulerian_path

[1] J. Edmonds, E. L. Johnson. Matching, Euler tours and the Chinese postman. Mathematical programming, Volume 5, Issue 1 (1973), 111-114.

[2] https://en.wikipedia.org/wiki/Eulerian_path

[3] http://web.math.princeton.edu/math_alive/5/Notes1.pdf

[1] Dinitz' Algorithm: The Original Version and Even's Version. 2006. Yefim Dinitz. In Theoretical Computer Science. Lecture Notes in Computer Science. Volume 3895. pp 218-240. https://doi.org/10.1007/11685654_10

[1] Boykov, Y., & Kolmogorov, V. (2004). An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. Pattern Analysis and Machine Intelligence, IEEE Transactions on, 26(9), 1124-1137. https://doi.org/10.1109/TPAMI.2004.60

[2] Vladimir Kolmogorov. Graph-based Algorithms for Multi-camera Reconstruction Problem. PhD thesis, Cornell University, CS Department, 2003. pp. 109-114. https://web.archive.org/web/20170809091249/https://pub.ist.ac.at/~vnk/papers/thesis.pdf

[1] Gusfield D: Very simple methods for all pairs network flow analysis. SIAM J Comput 19(1):143-155, 1990.

[1] Z. Kiraly, P. Kovacs. Efficient implementation of minimum-cost flow algorithms. Acta Universitatis Sapientiae, Informatica 4(1):67–118. 2012.

[2] R. Barr, F. Glover, D. Klingman. Enhancement of spanning tree labeling procedures for network optimization. INFOR 17(1):16–34. 1979.

[1] Shervashidze, Nino, Pascal Schweitzer, Erik Jan Van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler Lehman Graph Kernels. Journal of Machine Learning Research. 2011. http://www.jmlr.org/papers/volume12/shervashidze11a/shervashidze11a.pdf

[1] Shervashidze, Nino, Pascal Schweitzer, Erik Jan Van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler Lehman Graph Kernels. Journal of Machine Learning Research. 2011. http://www.jmlr.org/papers/volume12/shervashidze11a/shervashidze11a.pdf

[2] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu and Shantanu Jaiswa. graph2vec: Learning Distributed Representations of Graphs. arXiv. 2017 https://arxiv.org/pdf/1707.05005.pdf

[EG1960] Erdős and Gallai, Mat. Lapok 11 264, 1960.

[choudum1986] S.A. Choudum. "A simple proof of the Erdős-Gallai theorem on graph sequences." Bulletin of the Australian Mathematical Society, 33, pp 67-70, 1986. https://doi.org/10.1017/S0004972700002872

[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.

[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)

[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).

[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).

[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.

[1] A. Tripathi and S. Vijay. "A note on a theorem of Erdős & Gallai", Discrete Mathematics, 265, pp. 417-420 (2003).

[2] I.E. Zverovich and V.E. Zverovich. "Contributions to the theory of graphic sequences", Discrete Mathematics, 105, pp. 292-303 (1992).

[EG1960] Erdős and Gallai, Mat. Lapok 11 264, 1960.

[1] Luo, J.; Magee, C.L. (2011), Detecting evolving patterns of self-organizing networks by flow hierarchy measurement, Complexity, Volume 16 Issue 6 53-61. DOI: 10.1002/cplx.20368 http://web.mit.edu/~cmagee/www/documents/28-DetectingEvolvingPatterns_FlowHierarchy.pdf

[1] Chung, Fan and Linyuan Lu. "The Small World Phenomenon in Hybrid Power Law Graphs." *Complex Networks*. Springer Berlin Heidelberg, 2004. 89–104.

[1] Chung, Fan and Linyuan Lu. "The Small World Phenomenon in Hybrid Power Law Graphs." *Complex Networks*. Springer Berlin Heidelberg, 2004. 89–104.

[1] L. P. Cordella, P. Foggia, C. Sansone, M. Vento, "An Improved Algorithm for Matching Large Graphs", 3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition, Cuen,

pp. 149-159, 2001. https://www.researchgate.net/publication/200034365_An_Improved_Algorithm_for_Matching_Large_Graphs

[1] M. Houbraken, S. Demeyer, T. Michoel, P. Audenaert, D. Colle, M. Pickavet, "The Index-Based Subgraph Matching Algorithm with General Symmetries (ISMAGS): Exploiting Symmetry for Faster Subgraph Enumeration", PLoS One 9(5): e97896, 2014. https://doi.org/10.1371/journal.pone.0097896

[2] https://en.wikipedia.org/wiki/Maximum_common_induced_subgraph

[1] M. Houbraken, S. Demeyer, T. Michoel, P. Audenaert, D. Colle, M. Pickavet, "The Index-Based Subgraph Matching Algorithm with General Symmetries (ISMAGS): Exploiting Symmetry for Faster Subgraph Enumeration", PLoS One 9(5): e97896, 2014. https://doi.org/10.1371/journal.pone.0097896

[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

[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): 604-32, 1999. doi:10.1145/324133.324140. http://www.cs.cornell.edu/home/kleinber/auth.pdf.

[1] T. Zhou, L. Lu, Y.-C. Zhang. Predicting missing links via local information. Eur. Phys. J. B 71 (2009) 623. https://arxiv.org/pdf/0901.0553.pdf

[1] D. Liben-Nowell, J. Kleinberg. The Link Prediction Problem for Social Networks (2004). http://www.cs.cornell.edu/home/kleinber/link-pred.pdf

[1] D. Liben-Nowell, J. Kleinberg. The Link Prediction Problem for Social Networks (2004). http://www.cs.cornell.edu/home/kleinber/link-pred.pdf

[1] D. Liben-Nowell, J. Kleinberg. The Link Prediction Problem for Social Networks (2004). http://www.cs.cornell.edu/home/kleinber/link-pred.pdf

[1] Sucheta Soundarajan and John Hopcroft. Using community information to improve the precision of link prediction methods. In Proceedings of the 21st international conference companion on World Wide Web (WWW '12 Companion). ACM, New York, NY, USA, 607-608. http://doi.acm.org/10.1145/2187980.2188150

[1] Sucheta Soundarajan and John Hopcroft. Using community information to improve the precision of link prediction methods. In Proceedings of the 21st international conference companion on World Wide Web (WWW '12 Companion). ACM, New York, NY, USA, 607-608. http://doi.acm.org/10.1145/2187980.2188150

[1] Jorge Carlos Valverde-Rebaza and Alneu de Andrade Lopes. Link prediction in complex networks based on cluster information. In Proceedings of the 21st Brazilian conference on Advances in Artificial Intelligence (SBIA'12) https://doi.org/10.1007/978-3-642-34459-6_10

[1] Ahmad, I., Akhtar, M.U., Noor, S. et al. Missing Link Prediction using Common Neighbor and Centrality based Parameterized Algorithm. Sci Rep 10, 364 (2020). https://doi.org/10.1038/s41598-019-57304-y

[1] "Efficient Algorithms for Finding Maximum Matching in Graphs", Zvi Galil, ACM Computing Surveys, 1986.

[1] https://en.wikipedia.org/wiki/Graph_minor

[1] Patrick Doreian, Vladimir Batagelj, and Anuska Ferligoj. *Generalized Blockmodeling*. Cambridge University Press, 2004.

[1] Xiaowei Ying and Xintao Wu, On Randomness Measures for Social Networks, SIAM International Conference on Data Mining. 2009

[1] Wray L. Buntine. 1995. Chain graphs for learning. In Proceedings of the Eleventh conference on Uncertainty in artificial intelligence (UAI'95)

[1] J. A. Bondy, U. S. R. Murty, *Graph Theory*. Springer, 2008.

[1] Ulrik Brandes: The Left-Right Planarity Test 2009 http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.217.9208

[2] Takao Nishizeki, Md Saidur Rahman: Planar graph drawing Lecture Notes Series on Computing: Volume 12 2004

[1] M. Chrobak and T.H. Payne: A Linear-time Algorithm for Drawing a Planar Graph on a Grid 1989 http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.51.6677

[1] Y. Shi, M. Dehmer, X. Li, I. Gutman, "Graph Polynomials"

[1] M. Brandt, "The Tutte Polynomial." Talking About Combinatorial Objects Seminar, 2015 https://math.berkeley.edu/~brandtm/talks/tutte.pdf

[2] A. Björklund, T. Husfeldt, P. Kaski, M. Koivisto, "Computing the Tutte polynomial in vertex-exponential time" 49th Annual IEEE Symposium on Foundations of Computer Science, 2008 https://ieeexplore.ieee.org/abstract/document/4691000

[3] Y. Shi, M. Dehmer, X. Li, I. Gutman, "Graph Polynomials," p. 14

[4] Y. Shi, M. Dehmer, X. Li, I. Gutman, "Graph Polynomials," p. 46

[5] A. Nešetril, J. Goodall, "Graph invariants, homomorphisms, and the Tutte polynomial" https://iuuk.mff.cuni.cz/~andrew/Tutte.pdf

[6] D. B. West, "Introduction to Graph Theory," p. 84

[7] G. Coutinho, "A brief introduction to the Tutte polynomial" Structural Analysis of Complex Networks, 2011 https://homepages.dcc.ufmg.br/~gabriel/seminars/coutinho_tuttepolynomial_seminar.pdf

[8] J. A. Ellis-Monaghan, C. Merino, "Graph polynomials and their applications I: The Tutte polynomial" Structural Analysis of Complex Networks, 2011 https://arxiv.org/pdf/0803.3079.pdf

[1] D. B. West, "Introduction to Graph Theory," p. 222

[2] E. W. Weisstein "Chromatic Polynomial" MathWorld–A Wolfram Web Resource https://mathworld.wolfram.com/ChromaticPolynomial.html

[3] D. B. West, "Introduction to Graph Theory," p. 221

[4] J. Zhang, J. Goodall, "An Introduction to Chromatic Polynomials" https://math.mit.edu/~apost/courses/18.204_2018/Julie_Zhang_paper.pdf

[5] R. C. Read, "An Introduction to Chromatic Polynomials" Journal of Combinatorial Theory, 1968 https://math.berkeley.edu/~mrklug/ReadChromatic.pdf

[6] W. T. Tutte, "Graph-polynomials" Advances in Applied Mathematics, 2004 https://www.sciencedirect.com/science/article/pii/S0196885803000411

[7] R. P. Stanley, "Acyclic orientations of graphs" Discrete Mathematics, 2006 https://math.mit.edu/~rstan/pubs/pubfiles/18.pdf

[1] "An algorithm for computing simple k-factors.", Meijer, Henk, Yurai Núñez-Rodríguez, and David Rappaport, Information processing letters, 2009.

[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

[1] Zeina Abu-Aisheh, Romain Raveaux, Jean-Yves Ramel, Patrick Martineau. An Exact Graph Edit Distance Algorithm for Solving Pattern Recognition Problems. 4th International Conference on Pattern Recognition Applications and Methods 2015, Jan 2015, Lisbon, Portugal. 2015, <10.5220/0005209202710278>. <hal-01168816> https://hal.archives-ouvertes.fr/hal-01168816

[1] Zeina Abu-Aisheh, Romain Raveaux, Jean-Yves Ramel, Patrick Martineau. An Exact Graph Edit Distance Algorithm for Solving Pattern Recognition Problems. 4th International Conference on Pattern Recognition Applications and Methods 2015, Jan 2015, Lisbon, Portugal. 2015, <10.5220/0005209202710278>. <hal-01168816> https://hal.archives-ouvertes.fr/hal-01168816

[1] Zeina Abu-Aisheh, Romain Raveaux, Jean-Yves Ramel, Patrick Martineau. An Exact Graph Edit Distance Algorithm for Solving Pattern Recognition Problems. 4th International Conference on Pattern Recognition Applications and Methods 2015, Jan 2015, Lisbon, Portugal. 2015, <10.5220/0005209202710278>. <hal-01168816> https://hal.archives-ouvertes.fr/hal-01168816

[1] Zeina Abu-Aisheh, Romain Raveaux, Jean-Yves Ramel, Patrick Martineau. An Exact Graph Edit Distance Algorithm for Solving Pattern Recognition Problems. 4th International Conference on Pattern Recognition Applications and Methods 2015, Jan 2015, Lisbon, Portugal. 2015, <10.5220/0005209202710278>. <hal-01168816> https://hal.archives-ouvertes.fr/hal-01168816

[1] https://en.wikipedia.org/wiki/SimRank

[2] 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.

[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.

[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.

[1] R. Sedgewick, "Algorithms in C, Part 5: Graph Algorithms", Addison Wesley Professional, 3rd ed., 2001.

[1] R. Sedgewick, "Algorithms in C, Part 5: Graph Algorithms", Addison Wesley Professional, 3rd ed., 2001.

[1] Jin Y. Yen, "Finding the K Shortest Loopless Paths in a Network", Management Science, Vol. 17, No. 11, Theory Series (Jul., 1971), pp. 712-716.

[1] Small-world network:: https://en.wikipedia.org/wiki/Small-world_network

[1] Maslov, Sergei, and Kim Sneppen. "Specificity and stability in topology of protein networks." Science 296.5569 (2002): 910-913.

[1] Sporns, Olaf, and Jonathan D. Zwi. "The small world of the cerebral cortex." Neuroinformatics 2.2 (2004): 145-162.

[2] Maslov, Sergei, and Kim Sneppen. "Specificity and stability in topology of protein networks." Science 296.5569 (2002): 910-913.

[1] The brainstem reticular formation is a small-world, not scale-free, network M. D. Humphries, K. Gurney and T. J. Prescott, Proc. Roy. Soc. B 2006 273, 503-511, doi:10.1098/rspb.2005.3354.

[2] Humphries and Gurney (2008). "Network 'Small-World-Ness': A Quantitative Method for Determining Canonical Network Equivalence". PLoS One. 3 (4). PMID 18446219. doi:10.1371/journal.pone.0002051.

[1] Telesford, Joyce, Hayasaka, Burdette, and Laurienti (2011). "The Ubiquity of Small-World Networks". Brain Connectivity. 1 (0038): 367-75. PMC 3604768. PMID 22432451. doi:10.1089/brain.2011.0038.

[1] Lun Li, David Alderson, John C. Doyle, and Walter Willinger, Towards a Theory of Scale-Free Graphs: Definition, Properties, and Implications (Extended Version), 2005. https://arxiv.org/abs/cond-mat/0501169

[1] Burt, Ronald S. "Structural holes and good ideas". American Journal of Sociology (110): 349–399.

[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

[1] Burt, Ronald S. "Structural holes and good ideas". American Journal of Sociology (110): 349–399.

[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

[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.

[1] Erdős, Péter L., et al. "A Simple Havel-Hakimi Type Algorithm to Realize Graphical Degree Sequences of Directed Graphs." ArXiv:0905.4913 [Math], Jan. 2010. https://doi.org/10.48550/arXiv.0905.4913. Published 2010 in Elec. J. Combinatorics (17(1)). R66. http://www.combinatorics.org/Volume_17/PDF/v17i1r66.pdf

[2] "Combinatorics - Reaching All Possible Simple Directed Graphs with a given Degree Sequence with 2-Edge Swaps." Mathematics Stack Exchange, https://math.stackexchange.com/questions/22272/. Accessed 30 May 2022.

[1] C. Gkantsidis and M. Mihail and E. Zegura, The Markov chain simulation method for generating connected power law random graphs, 2003. http://citeseer.ist.psu.edu/gkantsidis03markov.html

[1] Threshold graphs: https://en.wikipedia.org/wiki/Threshold_graph

[1] Threshold graphs: https://en.wikipedia.org/wiki/Threshold_graph

[1] Tantau, Till. "A note on the complexity of the reachability problem for tournaments." *Electronic Colloquium on Computational Complexity*. 2001. <http://eccc.hpi-web.de/report/2001/092/>

[1] Tantau, Till. "A note on the complexity of the reachability problem for tournaments." *Electronic Colloquium on Computational Complexity*. 2001. <http://eccc.hpi-web.de/report/2001/092/>

[1] http://www.ics.uci.edu/~eppstein/PADS

[2] https://en.wikipedia.org/wiki/Depth-limited_search

[1] http://www.ics.uci.edu/~eppstein/PADS/BFS.py.

[2] https://en.wikipedia.org/wiki/Depth-limited_search

[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

[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

[1] Wang, Xiaodong, Lei Wang, and Yingjie Wu. "An optimal algorithm for Prufer codes." *Journal of Software Engineering and Applications* 2.02 (2009): 111. <https://doi.org/10.4236/jsea.2009.22016>

[1] Wang, Xiaodong, Lei Wang, and Yingjie Wu. "An optimal algorithm for Prufer codes." *Journal of Software Engineering and Applications* 2.02 (2009): 111. <https://doi.org/10.4236/jsea.2009.22016>

[1] V. Kulkarni, Generating random combinatorial objects, Journal of Algorithms, 11 (1990), pp. 185–207

[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

[1] Junction tree algorithm: https://en.wikipedia.org/wiki/Junction_tree_algorithm

[2] Finn V. Jensen and Frank Jensen. 1994. Optimal junction trees. In Proceedings of the Tenth international conference on Uncertainty in artificial intelligence (UAI'94). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 360–366.

[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

[1] Snijders, T. (2012). "Transitivity and triads." University of Oxford. https://web.archive.org/web/20170830032057/http://www.stats.ox.ac.uk/~snijders/Trans_Triads_ha.pdf

[1] Snijders, T. (2012). "Transitivity and triads." University of Oxford. https://web.archive.org/web/20170830032057/http://www.stats.ox.ac.uk/~snijders/Trans_Triads_ha.pdf

[1] Ulrik Brandes, Thomas Erlebach (eds.). *Network Analysis: Methodological Foundations*. Springer, 2005. <http://books.google.com/books?id=TTNhSm7HYrIC>

[1] Erwig, Martin. (2000), "The graph Voronoi diagram with applications." *Networks*, 36: 156–163. <dx.doi.org/10.1002/1097-0037(200010)36:3<156::AID-NET2>3.0.CO;2-L>

[1] Ronald C. Read and Robin J. Wilson, *An Atlas of Graphs*. Oxford University Press, 1998.

[atlas] Ronald C. Read and Robin J. Wilson, *An Atlas of Graphs*. Oxford University Press, 1998.

[1] An introduction to data structures and algorithms, James Andrew Storer, Birkhauser Boston 2001, (page 225).

[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.

[1] https://en.wikipedia.org/wiki/Bull_graph.

[1] https://en.wikipedia.org/wiki/Chv%C3%A1tal_graph

[2] https://mathworld.wolfram.com/ChvatalGraph.html

[1] https://en.wikipedia.org/wiki/Cube#Cubical_graph

[1] https://en.wikipedia.org/wiki/Desargues_graph

[2] https://mathworld.wolfram.com/DesarguesGraph.html

[1] https://mathworld.wolfram.com/DiamondGraph.html

[1] https://en.wikipedia.org/wiki/Regular_dodecahedron#Dodecahedral_graph

[2] https://mathworld.wolfram.com/DodecahedralGraph.html

[1] https://en.wikipedia.org/wiki/Frucht_graph

[2] https://mathworld.wolfram.com/FruchtGraph.html

[1] https://en.wikipedia.org/wiki/Heawood_graph

[2] https://mathworld.wolfram.com/HeawoodGraph.html

[3] https://www.win.tue.nl/~aeb/graphs/Heawood.html

[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

[1] https://mathworld.wolfram.com/HouseGraph.html

[1] https://mathworld.wolfram.com/HouseGraph.html

[1] https://mathworld.wolfram.com/IcosahedralGraph.html

[1] Krackhardt, David. "Assessing the Political Landscape: Structure, Cognition, and Power in Organizations". Administrative Science Quarterly. 35 (2): 342–369. doi:10.2307/2393394. JSTOR 2393394. June 1990.

[1] https://en.wikipedia.org/wiki/M%C3%B6bius%E2%80%93Kantor_graph

[1] https://mathworld.wolfram.com/OctahedralGraph.html

[2] https://en.wikipedia.org/wiki/Tur%C3%A1n_graph#Special_cases

[1] https://en.wikipedia.org/wiki/Pappus_graph

[1] https://en.wikipedia.org/wiki/Petersen_graph

[2] https://www.win.tue.nl/~aeb/drg/graphs/Petersen.html

[1] Figure 18.2, Chapter 18, Graph Algorithms (3rd Ed), Sedgewick

[1] https://en.wikipedia.org/wiki/Tetrahedron#Tetrahedral_graph

[1] https://en.wikipedia.org/wiki/Truncated_cube

[2] https://www.coolmath.com/reference/polyhedra-truncated-cube

[1] https://en.wikipedia.org/wiki/Truncated_tetrahedron

[1] https://en.wikipedia.org/wiki/Tutte_graph

[1] Vladimir Batagelj and Ulrik Brandes, "Efficient generation of large random networks", Phys. Rev. E, 71, 036113, 2005.

[1] P. Erdős and A. Rényi, On Random Graphs, Publ. Math. 6, 290 (1959).

[2] E. N. Gilbert, Random Graphs, Ann. Math. Stat., 30, 1141 (1959).

[1] Donald E. Knuth, The Art of Computer Programming, Volume 2/Seminumerical algorithms, Third Edition, Addison-Wesley, 1997.

[1] P. Erdős and A. Rényi, On Random Graphs, Publ. Math. 6, 290 (1959).

[2] E. N. Gilbert, Random Graphs, Ann. Math. Stat., 30, 1141 (1959).

[1] P. Erdős and A. Rényi, On Random Graphs, Publ. Math. 6, 290 (1959).

[2] E. N. Gilbert, Random Graphs, Ann. Math. Stat., 30, 1141 (1959).

[1] M. E. J. Newman and D. J. Watts, Renormalization group analysis of the small-world network model, Physics Letters A, 263, 341, 1999. https://doi.org/10.1016/S0375-9601(99)00757-4

[1] Duncan J. Watts and Steven H. Strogatz, Collective dynamics of small-world networks, Nature, 393, pp. 440–442, 1998.

[1] Duncan J. Watts and Steven H. Strogatz, Collective dynamics of small-world networks, Nature, 393, pp. 440–442, 1998.

[1] A. Steger and N. Wormald, Generating random regular graphs quickly, Probability and Computing 8 (1999), 377-396, 1999. http://citeseer.ist.psu.edu/steger99generating.html

[2] Jeong Han Kim and Van H. Vu, Generating random regular graphs, Proceedings of the thirty-fifth ACM symposium on Theory of computing, San Diego, CA, USA, pp 213–222, 2003. http://portal.acm.org/citation.cfm?id=780542.780576

[1] A. L. Barabási and R. Albert "Emergence of scaling in random networks", Science 286, pp 509-512, 1999.

[1] N. Moshiri "The dual-Barabasi-Albert model", arXiv:1810.10538.

[1] Albert, R., & Barabási, A. L. (2000) Topology of evolving networks: local events and universality Physical review letters, 85(24), 5234.

[1] P. Holme and B. J. Kim, "Growing scale-free networks with tunable clustering", Phys. Rev. E, 65, 026107, 2002.

[1] Bollobás, Béla, Janson, S. and Riordan, O. "The phase transition in inhomogeneous random graphs", *Random Structures Algorithms*, 31, 3–122, 2007.

[2] Hagberg A, Lemons N (2015), "Fast Generation of Sparse Random Kernel Graphs". PLoS ONE 10(9): e0135177, 2015. doi:10.1371/journal.pone.0135177

[1] I. Ispolatov, P. L. Krapivsky, A. Yuryev, "Duplication-divergence model of protein interaction network", Phys. Rev. E, 71, 061911, 2005.

[1] Knudsen Michael, and Carsten Wiuf. "A Markov chain approach to randomly grown graphs." Journal of Applied Mathematics 2008. <https://doi.org/10.1155/2008/190836>

[1] M.E.J. Newman, "The structure and function of complex networks", SIAM REVIEW 45-2, pp 167-256, 2003.

[1] Newman, M. E. J. and Strogatz, S. H. and Watts, D. J. Random graphs with arbitrary degree distributions and their applications Phys. Rev. E, 64, 026118 (2001)

[1] Fan Chung and L. Lu, Connected components in random graphs with given expected degree sequences, Ann. Combinatorics, 6, pp. 125-145, 2002.

[2] Joel Miller and Aric Hagberg, Efficient generation of networks with given expected degrees, in Algorithms and Models for the Web-Graph (WAW 2011), Alan Frieze, Paul Horn, and Paweł Prałat (Eds), LNCS 6732, pp. 115-126, 2011.

[1] Hakimi S., On Realizability of a Set of Integers as Degrees of the Vertices of a Linear Graph. I, Journal of SIAM, 10(3), pp. 496-506 (1962)

[2] Kleitman D.J. and Wang D.L. Algorithms for Constructing Graphs and Digraphs with Given Valences and Factors Discrete Mathematics, 6(1), pp. 79-88 (1973)

[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)

[1] Moshen Bayati, Jeong Han Kim, and Amin Saberi, A sequential algorithm for generating random graphs. Algorithmica, Volume 58, Number 4, 860-910, DOI: 10.1007/s00453-009-9340-1

[1] Joel C. Miller. "Percolation and epidemics in random clustered networks". In: Physical review. E, Statistical, nonlinear, and soft matter physics 80 (2 Part 1 August 2009).

[2] M. E. J. Newman. "Random Graphs with Clustering". In: Physical Review Letters 103 (5 July 2009)

[1] P. L. Krapivsky and S. Redner, Organization of Growing Random Networks, Phys. Rev. E, 63, 066123, 2001.

[1] P. L. Krapivsky and S. Redner, Organization of Growing Random Networks, Phys. Rev. E, 63, 066123, 2001.

[1] P. L. Krapivsky and S. Redner, Network Growth by Copying, Phys. Rev. E, 71, 036118, 2005k.},

[1] B. Bollobás, C. Borgs, J. Chayes, and O. Riordan, Directed scale-free graphs, Proceedings of the fourteenth annual ACM-SIAM Symposium on Discrete Algorithms, 132–139, 2003.

[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

[1] J. Kleinberg. The small-world phenomenon: An algorithmic perspective. Proc. 32nd ACM Symposium on Theory of Computing, 2000.

[1] Penrose, Mathew, *Random Geometric Graphs*, Oxford Studies in Probability, 5, 2003.

[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

[1] http://cole-maclean.github.io/blog/files/thesis.pdf

[1] B. M. Waxman, *Routing of multipoint connections*. IEEE J. Select. Areas Commun. 6(9),(1988) 1617–1622.

[1] A. Elmokashfi, A. Kvalbein and C. Dovrolis, "On the Scalability of BGP: The Role of Topology Growth," in IEEE Journal on Selected Areas in Communications, vol. 28, no. 8, pp. 1250-1261, October 2010.

[1] K.B. Singer-Cohen, Random Intersection Graphs, 1995, PhD thesis, Johns Hopkins University

[2] Fill, J. A., Scheinerman, E. R., and Singer-Cohen, K. B., Random intersection graphs when m = !(n): An equivalence theorem relating the evolution of the g(n, m, p) and g(n, p) models. Random Struct. Algorithms 16, 2 (2000), 156–176.

[1] Godehardt, E., and Jaworski, J. Two models of random intersection graphs and their applications. Electronic Notes in Discrete Mathematics 10 (2001), 129–132.

[1] Nikoletseas, S. E., Raptopoulos, C., and Spirakis, P. G. The existence and efficient construction of large independent sets in general random intersection graphs. In ICALP (2004), J. D´ıaz, J. Karhum¨aki, A. Lepist¨o, and D. Sannella, Eds., vol. 3142 of Lecture Notes in Computer Science, Springer, pp. 1029–1040.

[1] Zachary, Wayne W. "An Information Flow Model for Conflict and Fission in Small Groups." *Journal of Anthropological Research*, 33, 452–473, (1977).

[1] A. Davis, Gardner, B. B., Gardner, M. R., 1941. Deep South. University of Chicago Press, Chicago, IL.

[1] Ronald L. Breiger and Philippa E. Pattison Cumulated social roles: The duality of persons and their algebras,1 Social Networks, Volume 8, Issue 3, September 1986, Pages 215-256

[1] D. E. Knuth, 1993. The Stanford GraphBase: a platform for combinatorial computing, pp. 74-87. New York: AcM Press.

[1] Watts, D. J. 'Networks, Dynamics, and the Small-World Phenomenon.' Amer. J. Soc. 105, 493-527, 1999.

[1] Watts, D. J. 'Networks, Dynamics, and the Small-World Phenomenon.' Amer. J. Soc. 105, 493-527, 1999.

[1] Ulrik Brandes, Marco Gaertler, Dorothea Wagner, Experiments on Graph Clustering Algorithms, In the proceedings of the 11th Europ. Symp. Algorithms, 2003.

[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

[1] A. Condon, R.M. Karp, Algorithms for graph partitioning on the planted partition model, Random Struct. Algor. 18 (2001) 116-140.

[2] Santo Fortunato 'Community Detection in Graphs' Physical Reports Volume 486, Issue 3-5 p. 75-174. https://arxiv.org/abs/0906.0612

[1] Santo Fortunato 'Community Detection in Graphs' Physical Reports Volume 486, Issue 3-5 p. 75-174. https://arxiv.org/abs/0906.0612

[1] Santo Fortunato, Community Detection in Graphs, Physics Reports Volume 486, Issues 3-5, February 2010, Pages 75-174. https://arxiv.org/abs/0906.0612

[1] Holland, P. W., Laskey, K. B., & Leinhardt, S., "Stochastic blockmodels: First steps", Social networks, 5(2), 109-137, 1983.

[1] M. Gjoka, M. Kurant, A. Markopoulou, "2.5K Graphs: from Sampling to Generation", IEEE Infocom, 2013.

[2] I. Stanton, A. Pinar, "Constructing and sampling graphs with a prescribed joint degree distribution", Journal of Experimental Algorithmics, 2012.

[H] Harary, F. "The Maximum Connectivity of a Graph." Proc. Nat. Acad. Sci. USA 48, 1142-1146, 1962.

[1] F. T. Boesch, A. Satyanarayana, and C. L. Suffel, "A Survey of Some Network Reliability Analysis and Synthesis Results," Networks, pp. 99-107, 2009.

[2] Harary, F. "The Maximum Connectivity of a Graph." Proc. Nat. Acad. Sci. USA 48, 1142-1146, 1962.

[1] Weisstein, Eric W. "Harary Graph." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/HararyGraph.html.

[2] Harary, F. "The Maximum Connectivity of a Graph." Proc. Nat. Acad. Sci. USA 48, 1142-1146, 1962.

[0] D.G. Corneil, H. Lerchs, L.Stewart Burlingham, "Complement reducible graphs", Discrete Applied Mathematics, Volume 3, Issue 3, 1981, Pages 163-174, ISSN 0166-218X.

[1] D.G. Corneil, H. Lerchs, L.Stewart Burlingham, "Complement reducible graphs", Discrete Applied Mathematics, Volume 3, Issue 3, 1981, Pages 163-174, ISSN 0166-218X.

[1] Herzberg, A. M., & Murty, M. R. (2007). Sudoku squares and chromatic polynomials. Notices of the AMS, 54(6), 708-717.

[2] Sander, Torsten (2009), "Sudoku graphs are integral", Electronic Journal of Combinatorics, 16 (1): Note 25, 7pp, MR 2529816

[3] Wikipedia contributors. "Glossary of Sudoku." Wikipedia, The Free Encyclopedia, 3 Dec. 2019. Web. 22 Dec. 2019.

[1] Herzberg, A. M., & Murty, M. R. (2007). Sudoku squares and chromatic polynomials. Notices of the AMS, 54(6), 708-717.

[2] Sander, Torsten (2009), "Sudoku graphs are integral", Electronic Journal of Combinatorics, 16 (1): Note 25, 7pp, MR 2529816

[3] Wikipedia contributors. "Glossary of Sudoku." Wikipedia, The Free Encyclopedia, 3 Dec. 2019. Web. 22 Dec. 2019.

[1] Gil Strang, Network applications: A = incidence matrix, http://videolectures.net/mit18085f07_strang_lec03/

[1] Fan Chung-Graham, Spectral Graph Theory, CBMS Regional Conference Series in Mathematics, Number 92, 1997.

[2] Steve Butler, Interlacing For Weighted Graphs Using The Normalized Laplacian, Electronic Journal of Linear Algebra, Volume 16, pp. 90-98, March 2007.

[1] Fan Chung (2005). Laplacians and the Cheeger inequality for directed graphs. Annals of Combinatorics, 9(1), 2005

[1] Fan Chung (2005). Laplacians and the Cheeger inequality for directed graphs. Annals of Combinatorics, 9(1), 2005

[1] A. Saade, F. Krzakala and L. Zdeborová "Spectral Clustering of Graphs with the Bethe Hessian", Advances in Neural Information Processing Systems, 2014.

[2] C. M. Le, E. Levina "Estimating the number of communities in networks by spectral methods" arXiv:1507.00827, 2015.

[1] M. E. J. Newman, "Modularity and community structure in networks", Proc. Natl. Acad. Sci. USA, vol. 103, pp. 8577-8582, 2006.

[1] E. A. Leicht, M. E. J. Newman, "Community structure in directed networks", Phys. Rev Lett., vol. 100, no. 11, p. 118703, 2008.

[1] A. Saade, F. Krzakala and L. Zdeborová "Spectral clustering of graphs with the bethe hessian", Advances in Neural Information Processing Systems. 2014.

[1] M. E. J. Newman, "Modularity and community structure in networks", Proc. Natl. Acad. Sci. USA, vol. 103, pp. 8577-8582, 2006.

[1] Scipy Dev. References, "Sparse Matrices", https://docs.scipy.org/doc/scipy/reference/sparse.html

[1] GEXF File Format, http://gexf.net/

[1] GEXF File Format, http://gexf.net/

[2] GEXF schema, http://gexf.net/schema.html

[1] GEXF File Format, https://gephi.org/gexf/format/

[1] Cytoscape user's manual: http://manual.cytoscape.org/en/stable/index.html

[1] Cytoscape user's manual: http://manual.cytoscape.org/en/stable/index.html

[1] http://www.algorithmic-solutions.info/leda_guide/graphs/leda_native_graph_fileformat.html

[1] http://www.algorithmic-solutions.info/leda_guide/graphs/leda_native_graph_fileformat.html

[1] Graph6 specification <http://users.cecs.anu.edu.au/~bdm/data/formats.html>

[1] Graph6 specification <http://users.cecs.anu.edu.au/~bdm/data/formats.html>

[1] Graph6 specification <http://users.cecs.anu.edu.au/~bdm/data/formats.html>

[1] Graph6 specification <http://users.cecs.anu.edu.au/~bdm/data/formats.html>

[1] Sparse6 specification <https://users.cecs.anu.edu.au/~bdm/data/formats.html>

[1] Sparse6 specification <https://users.cecs.anu.edu.au/~bdm/data/formats.html>

[1] Graph6 specification <https://users.cecs.anu.edu.au/~bdm/data/formats.html>

[1] Sparse6 specification <https://users.cecs.anu.edu.au/~bdm/data/formats.html>

[1] Harary, F. and Read, R. "Is the Null Graph a Pointless Concept?" In Graphs and Combinatorics Conference, George Washington University. New York: Springer-Verlag, 1973.

[1] Luc Devroye, Non-Uniform Random Variate Generation, Springer-Verlag, New York, 1986.

[1] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices, In Proc. 24th Nat. Conf. ACM, pages 157-172, 1969. http://doi.acm.org/10.1145/800195.805928

[2] Steven S. Skiena. 1997. The Algorithm Design Manual. Springer-Verlag New York, Inc., New York, NY, USA.

[1] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices, In Proc. 24th Nat. Conf. ACM, pages 157-72, 1969. http://doi.acm.org/10.1145/800195.805928

[2] Steven S. Skiena. 1997. The Algorithm Design Manual. Springer-Verlag New York, Inc., New York, NY, USA.

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2001). Introduction to algorithms second edition.

[2] Knuth, D. E. (1997). The art of computer programming (Vol. 3). Pearson Education.

PYTHON MODULE INDEX

a

c

d

e

g

l

r

u

INDEX

Non-alphabetical

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z