NetworkX Reference

Release 1.5

Aric Hagberg, Dan Schult, Pieter Swart

June 04, 2011

CONTENTS

CHAPTER

ONE

INTRODUCTION

NetworkX is a Python-based package for the creation, manipulation, and study of the structure, dynamics, and function of complex networks.

The structure of a graph or network is encoded in the **edges** (connections, links, ties, arcs, bonds) between **nodes** (vertices, sites, actors). If unqualified, by graph we mean an undirected graph, i.e. no multiple edges are allowed. By a network we usually mean a graph with weights (fields, properties) on nodes and/or edges.

1.1 Who uses NetworkX?

The potential audience for NetworkX includes mathematicians, physicists, biologists, computer scientists, and social scientists. The current state of the art of the science of complex networks is presented in Albert and Barabási [BA02], Newman [Newman03], and Dorogovtsev and Mendes [DM03]. See also the classic texts [Bollobas01], [Diestel97] and [West01] for graph theoretic results and terminology. For basic graph algorithms, we recommend the texts of Sedgewick, e.g. [Sedgewick01] and [Sedgewick02] and the survey of Brandes and Erlebach [BE05].

1.2 The Python programming language

Why Python? Past experience showed this approach to maximize productivity, power, multi-disciplinary scope (applications include large communication, social, data and biological networks), and platform independence. This philosophy does not exclude using whatever other language is appropriate for a specific subtask, since Python is also an excellent "glue" language [Langtangen04]. Equally important, Python is free, well-supported and a joy to use. Among the many guides to Python, we recommend the documentation at http://www.python.org and the text by Alex Martelli [Martelli03].

1.3 Free software

NetworkX is free software; you can redistribute it and/or modify it under the terms of the *NetworkX License*. We welcome contributions from the community. Information on NetworkX development is found at the NetworkX Developer Zone https://networkx.lanl.gov/trac.

1.4 Goals

NetworkX is intended to:

- Be a tool to study the structure and dynamics of social, biological, and infrastructure networks
- Provide ease-of-use and rapid development in a collaborative, multidisciplinary environment
- Be an Open-source software package that can provide functionality to a diverse community of active and easily participating users and developers.
- Provide an easy interface to existing code bases written in C, C++, and FORTRAN
- Painlessly slurp in large nonstandard data sets
- Provide a standard API and/or graph implementation that is suitable for many applications.

1.5 History

- NetworkX was inspired by Guido van Rossum's 1998 Python graph representation essay [vanRossum98].
- First public release in April 2005. Version 1.0 released in 2009.

1.5.1 What Next

- `A Brief Tour`
- `Installing`
- `Reference`
- `Examples`

CHAPTER

TWO

OVERVIEW

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 analyse 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).

2.1 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. 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 adjaceny 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.

2.1.1 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 tricky users could design edge data objects 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*

2.2 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 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 to get the weight for each edge.

Other attributes can be assigned to an edge by using keyword/value pairs when adding edges. You can use any keyword except 'weight' to name your attribute and can then easily query the edge data by 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.

2.2.1 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=[('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/index` 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 powerlaw_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.

2.2.2 Graph Reporting

Class methods are used for the basic reporting functions neighbors, edges and degree. Reporting of lists is often needed only to iterate through that list so we supply iterator versions of many property reporting methods. For example edges() and nodes() have corresponding methods edges_iter() and nodes_iter(). Using these methods when you can will save memory and often time as well.

The basic graph relationship of an edge can be obtained in two basic ways. One can look for neighbors of a node or one can look for edges incident to a node. 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 avoidance of notation like G[u,v] in favor of G[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 each edge while G.neighbors(n) or G[n] is slightly faster but doesn't remove duplicates.

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*.

2.2.3 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 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']
```

2.2.4 Drawing

While NetworkX is not designed as a network layout 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 in a dictionary or computed with a layout function. The edges are then lines between those dots.

```
>>> G=nx.cubical_graph()
>>> nx.draw(G)    # default spring_layout
>>> nx.draw(G,pos=nx.spectral_layout(G), nodecolor='r',edge_color='b')
```

See the `examples` for more ideas.

2.2.5 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. The expression G[u][v] returns the edge attribute dictionary itself. A dictionary of lists would have also been possible, but not allowed 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'), ('B','C')

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

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 and one for predecessors. 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 use a dictionary of attributes for each edge. We use a dict-of-dicts-of-dicts data structure with the inner dictionary storing "name-value" relationships for that edge.

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

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

CHAPTER

THREE

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.

3.1 Which graph class should I use?

Graph Type	NetworkX Class
Undirected Simple	Graph
Directed Simple	DiGraph
With Self-loops	Graph, DiGraph
With Parallel edges	MultiGraph, MultiDiGraph

3.2 Basic graph types

3.2.1 Graph – Undirected graphs with self loops

Overview

networkx.**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.

Edges are represented as links between nodes with optional key/value attributes.

Parameters **data** : input graph

Data to initialize graph. If 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 NumPy matrix or 2d ndarray, a SciPy sparse matrix, or a 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.Graph()
>>> H.add_path([0,1,2,3,4,5,6,7,8,9])
>>> 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.node

```
>>> G.add_node(1, time='5pm')
>>> G.add_nodes_from([3], time='2pm')
>>> G.node[1]
{'time': '5pm'}
>>> G.node[1]['room'] = 714
>>> G.nodes(data=True)
[(1, {'room': 714, 'time': '5pm'}), (3, {'time': '2pm'})]
```

Warning: adding a node to G.node does not add it to the graph.

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

```
>>> 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.edge[1][2]['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 keyed by neighbor to edge attributes
...            # Note: you should not change this dict manually!
{2: {'color': 'blue', 'weight': 4}}
```

The fastest way to traverse all edges of a graph is via adjacency_iter(), but the edges() method is often more convenient.

```
>>> for n,nbrsdict in G.adjacency_iter():
...     for nbr,eattr in nbrsdict.items():
...        if 'weight' in eattr:
...            (n,nbr,eattr['weight'])
(1, 2, 4)
(2, 1, 4)
(2, 3, 8)
(3, 2, 8)
>>> [ (u,v,edata['weight']) for u,v,edata in G.edges(data=True) if 'weight' in edata ]
[(1, 2, 4), (2, 3, 8)]
```

Reporting:

Simple graph information is obtained using methods. Iterator versions of many reporting methods exist for efficiency. Methods exist for reporting nodes(), edges(), neighbors() and degree() as well as the number of nodes and edges.

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

Adding and removing nodes and edges

Graph.__init__(**attr[, data])	Initialize a graph with edges, name, graph attributes.
Graph.add_node(n, **attr[, attr_dict])	Add a single node n and update node attributes.
Graph.add_nodes_from(nodes, **attr)	Add multiple nodes.
Graph.remove_node(n)	Remove node n.
Graph.remove_nodes_from(nodes)	Remove multiple nodes.
Graph.add_edge(u, v, **attr[, attr_dict])	Add an edge between u and v.
Graph.add_edges_from(ebunch, **attr[, attr_dict])	Add all the edges in ebunch.
Graph.add_weighted_edges_from(ebunch, **attr)	Add all the edges in ebunch as weighted edges with specified weights.
Graph.remove_edge(u, v)	Remove the edge between u and v.
Graph.remove_edges_from(ebunch)	Remove all edges specified in ebunch.
Graph.add_star(nodes, **attr)	Add a star.
Graph.add_path(nodes, **attr)	Add a path.
Graph.add_cycle(nodes, **attr)	Add a cycle.
Graph.clear()	Remove all nodes and edges from the graph.

networkx.Graph.__init__

Graph.**__init__**(*data=None*, ***attr*)

Initialize a graph with edges, name, graph attributes.

Parameters **data** : input graph

Data to initialize graph. If 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 NumPy matrix or 2d ndarray, a SciPy sparse matrix, or a PyGraphviz graph.

name : string, optional (default='')

An optional name for the 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'}
```

networkx.Graph.add_node

Graph.**add_node**(*n*, *attr_dict=None*, ***attr*)

Add a single node n and update node attributes.

Parameters **n** : node

A node can be any hashable Python object except None.

attr_dict : dictionary, optional (default= no attributes)

Dictionary of node attributes. Key/value pairs will update existing data associated with the node.

attr : keyword arguments, optional

Set or change 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))
```

networkx.Graph.add_nodes_from

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

Add multiple nodes.

Parameters **nodes** : 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 generally.

See Also:

add_node

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.node[1]['size']
11
>>> H = nx.Graph()
>>> H.add_nodes_from(G.nodes(data=True))
>>> H.node[1]['size']
11
```

networkx.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.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> G.edges()
[(0, 1), (1, 2)]
>>> G.remove_node(1)
```

```
>>> G.edges()
[]
```

networkx.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

Examples

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

networkx.Graph.add_edge

Graph.**add_edge**(*u*, *v*, *attr_dict=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 providing a dictionary with key/value pairs. See examples below.

Parameters **u,v** : nodes

Nodes can be, for example, strings or numbers. Nodes must be hashable (and not None) Python objects.

attr_dict : dictionary, optional (default= no attributes)

Dictionary of edge attributes. Key/value pairs will update existing data associated with the edge.

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.

NetworkX algorithms designed for weighted graphs use as the edge weight a numerical value assigned to the keyword 'weight'.

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)
```

networkx.Graph.add_edges_from

Graph.**add_edges_from**(*ebunch*, *attr_dict=None*, ***attr*)

Add all the edges in ebunch.

Parameters **ebunch** : container of edges

Each edge given in the container will be added to the graph. The edges must be given as as 2-tuples (u,v) or 3-tuples (u,v,d) where d is a dictionary containing edge data.

attr_dict : dictionary, optional (default= no attributes)

Dictionary of edge attributes. Key/value pairs will update existing data associated with each edge.

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.

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')
```

networkx.Graph.add_weighted_edges_from

Graph.**add_weighted_edges_from**(*ebunch*, ***attr*)

Add all the edges in ebunch as weighted edges with specified weights.

Parameters **ebunch** : 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.

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 has no effect but any edge data will be updated when each duplicate edge is added.

Examples

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

networkx.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.Graph()    # or DiGraph, etc
>>> G.add_path([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
```

networkx.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.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> ebunch=[(1,2),(2,3)]
>>> G.remove_edges_from(ebunch)
```

networkx.Graph.add_star

Graph.**add_star**(*nodes*, ***attr*)

Add a star.

The first node in nodes is the middle of the star. It is connected to all other nodes.

Parameters **nodes** : 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()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_star([0,1,2,3])
>>> G.add_star([10,11,12],weight=2)
```

networkx.Graph.add_path

Graph.**add_path**(*nodes*, ***attr*)

Add a path.

Parameters **nodes** : 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()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.add_path([10,11,12],weight=7)
```

networkx.Graph.add_cycle

Graph.**add_cycle**(*nodes*, ***attr*)

Add a cycle.

Parameters **nodes: 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
>>> G.add_cycle([0,1,2,3])
>>> G.add_cycle([10,11,12],weight=7)
```

networkx.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.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.clear()
>>> G.nodes()
[]
>>> G.edges()
[]
```

Iterating over nodes and edges

Graph.nodes([data])	Return a list of the nodes in the graph.
Graph.nodes_iter([data])	Return an iterator over the nodes.
Graph.__iter__()	Iterate over the nodes.
Graph.edges([nbunch, data])	Return a list of edges.
Graph.edges_iter([nbunch, data])	Return an iterator over the edges.
Graph.get_edge_data(u, v[, default])	Return the attribute dictionary associated with edge (u,v).
Graph.neighbors(n)	Return a list of the nodes connected to the node n.
Graph.neighbors_iter(n)	Return an iterator over all neighbors of node n.
Graph.__getitem__(n)	Return a dict of neighbors of node n.
Graph.adjacency_list()	Return an adjacency list representation of the graph.
Graph.adjacency_iter()	Return an iterator of (node, adjacency dict) tuples for all nodes.
Graph.nbunch_iter([nbunch])	Return an iterator of nodes contained in nbunch that are also in the graph.

networkx.Graph.nodes

Graph.**nodes**(*data=False*)

Return a list of the nodes in the graph.

Parameters **data** : boolean, optional (default=False)

If False return a list of nodes. If True return a two-tuple of node and node data dictionary

Returns **nlist** : list

A list of nodes. If data=True a list of two-tuples containing (node, node data dictionary).

Examples

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> G.nodes()
[0, 1, 2]
>>> G.add_node(1, time='5pm')
>>> G.nodes(data=True)
[(0, {}), (1, {'time': '5pm'}), (2, {})]
```

networkx.Graph.nodes_iter

Graph.**nodes_iter**(*data=False*)

Return an iterator over the nodes.

Parameters **data** : boolean, optional (default=False)

If False the iterator returns nodes. If True return a two-tuple of node and node data dictionary

Returns **niter** : iterator

An iterator over nodes. If data=True the iterator gives two-tuples containing (node, node data, dictionary)

Notes

If the node data is not required it is simpler and equivalent to use the expression 'for n in G'.

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

Examples

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

>>> [d for n,d in G.nodes_iter(data=True)]
[{}, {}, {}]
```

networkx.Graph.__iter__

Graph.**__iter__**()

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

Returns **niter** : iterator

An iterator over all nodes in the graph.

Examples

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

networkx.Graph.edges

Graph.**edges**(*nbunch=None*, *data=False*)

Return a list of edges.

Edges are returned as tuples with optional data in the order (node, neighbor, data).

Parameters **nbunch** : iterable container, optional (default= all nodes)

A container of nodes. The container will be iterated through once.

data : bool, optional (default=False)

Return two tuples (u,v) (False) or three-tuples (u,v,data) (True).

Returns **edge_list: list of edge tuples** :

Edges that are adjacent to any node in nbunch, or a list of all edges if nbunch is not specified.

See Also:

edges_iter return an iterator over the edges

Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored.

Examples

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.edges()
[(0, 1), (1, 2), (2, 3)]
>>> G.edges(data=True) # default edge data is {} (empty dictionary)
[(0, 1, {}), (1, 2, {}), (2, 3, {})]
>>> G.edges([0,3])
[(0, 1), (3, 2)]
>>> G.edges(0)
[(0, 1)]
```

networkx.Graph.edges_iter

Graph.**edges_iter**(*nbunch=None*, *data=False*)

Return an iterator over the edges.

Edges are returned as tuples with optional data in the order (node, neighbor, data).

Parameters **nbunch** : iterable container, optional (default= all nodes)

A container of nodes. The container will be iterated through once.

data : bool, optional (default=False)

If True, return edge attribute dict in 3-tuple (u,v,data).

Returns **edge_iter** : iterator

An iterator of (u,v) or (u,v,d) tuples of edges.

See Also:

`edges` return a list of edges

Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored.

Examples

```
>>> G = nx.Graph()   # or MultiGraph, etc
>>> G.add_path([0,1,2,3])
>>> [e for e in G.edges_iter()]
[(0, 1), (1, 2), (2, 3)]
>>> list(G.edges_iter(data=True)) # default data is {} (empty dict)
[(0, 1, {}), (1, 2, {}), (2, 3, {})]
>>> list(G.edges_iter([0,3]))
[(0, 1), (3, 2)]
>>> list(G.edges_iter(0))
[(0, 1)]
```

networkx.Graph.get_edge_data

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

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

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.

Notes

It is faster to use G[u][v].

```
>>> G = nx.Graph()   # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G[0][1]
{}
```

Warning: Assigning G[u][v] corrupts the graph data structure. But it is safe to assign attributes to that dictionary,

```
>>> G[0][1]['weight'] = 7
>>> G[0][1]['weight']
7
>>> G[1][0]['weight']
7
```

Examples

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

networkx.Graph.neighbors

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

Return a list of the nodes connected to the node n.

Parameters **n** : node

A node in the graph

Returns **nlist** : list

A list of nodes that are adjacent to n.

Raises **NetworkXError** :

If the node n is not in the graph.

Notes

It is usually more convenient (and faster) to access the adjacency dictionary as G[n]:

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge('a','b',weight=7)
>>> G['a']
{'b': {'weight': 7}}
```

Examples

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

networkx.Graph.neighbors_iter

Graph.**neighbors_iter**(*n*)
Return an iterator over all neighbors of node n.

Notes

It is faster to use the idiom "in G[0]", e.g.

```
>>> G = nx.path_graph(4)
>>> [n for n in G[0]]
[1]
```

Examples

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

networkx.Graph.__getitem__

Graph.**__getitem__**(*n*)
Return a dict of neighbors of node n. Use the expression '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 similar to G.neighbors(n) but the internal data dictionary is returned instead of a list.

Assigning G[n] will corrupt the internal graph data structure. Use G[n] for reading data only.

Examples

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G[0]
{1: {}}
```

networkx.Graph.adjacency_list

Graph.**adjacency_list**()
Return an adjacency list representation of the graph.

The output adjacency list is in the order of G.nodes(). For directed graphs, only outgoing adjacencies are included.

Returns **adj_list** : lists of lists

The adjacency structure of the graph as a list of lists.

See Also:

adjacency_iter

Examples

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.adjacency_list() # in order given by G.nodes()
[[1], [0, 2], [1, 3], [2]]
```

networkx.Graph.adjacency_iter

Graph.**adjacency_iter**()

Return an iterator of (node, adjacency dict) tuples for all nodes.

This is the fastest way to look at every edge. For directed graphs, only outgoing adjacencies are included.

Returns **adj_iter** : iterator

An iterator of (node, adjacency dictionary) for all nodes in the graph.

See Also:

adjacency_list

Examples

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

networkx.Graph.nbunch_iter

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

Return an iterator of 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** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

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 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.

Information about graph structure

Graph.has_node(n)	Return True if the graph contains the node n.
Graph.__contains__(n)	Return True if n is a node, False otherwise. Use the expression
Graph.has_edge(u, v)	Return True if the edge (u,v) is in the graph.
Graph.order()	Return the number of nodes in the graph.
Graph.number_of_nodes()	Return the number of nodes in the graph.
Graph.__len__()	Return the number of nodes.
Graph.degree([nbunch, weighted])	Return the degree of a node or nodes.
Graph.degree_iter([nbunch, weighted])	Return an iterator for (node, degree).
Graph.size([weighted])	Return the number of edges.
Graph.number_of_edges([u, v])	Return the number of edges between two nodes.
Graph.nodes_with_selfloops()	Return a list of nodes with self loops.
Graph.selfloop_edges([data])	Return a list of selfloop edges.
Graph.number_of_selfloops()	Return the number of selfloop edges.

networkx.Graph.has_node

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

Return True if the graph contains the node n.

Parameters **n** : node

Examples

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> G.has_node(0)
True
```

It is more readable and simpler to use

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

networkx.Graph.__contains__

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

Return True if n is a node, False otherwise. Use the expression 'n in G'.

Examples

```
>>> G = nx.Graph()   # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> 1 in G
True
```

networkx.Graph.has_edge

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

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

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

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

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

The following syntax are all equivalent:

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

networkx.Graph.order

Graph.**order**()

Return the number of nodes in the graph.

Returns **nnodes** : int

The number of nodes in the graph.

See Also:

number_of_nodes, __len__

networkx.Graph.number_of_nodes

Graph.**number_of_nodes**()

Return the number of nodes in the graph.

Returns **nnodes** : int

The number of nodes in the graph.

See Also:

order, __len__

Examples

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

networkx.Graph.__len__

Graph.**__len__**()

Return the number of nodes. Use the expression 'len(G)'.

Returns **nnodes** : int

The number of nodes in the graph.

Examples

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> len(G)
4
```

networkx.Graph.degree

Graph.**degree**(*nbunch=None*, *weighted=False*)

Return the degree of a node or nodes.

The node degree is the number of edges adjacent to that node.

Parameters **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

weighted : bool, optional (default=False)

If True return the sum of edge weights adjacent to the node.

Returns **nd** : dictionary, or number

A dictionary with nodes as keys and degree as values or a number if a single node is specified.

Examples

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

networkx.Graph.degree_iter

Graph.**degree_iter**(*nbunch=None*, *weighted=False*)

Return an iterator for (node, degree).

The node degree is the number of edges adjacent to the node.

Parameters **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

weighted : bool, optional (default=False)

If True return the sum of edge weights adjacent to the node.

Returns **nd_iter** : an iterator

The iterator returns two-tuples of (node, degree).

See Also:

degree

Examples

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

networkx.Graph.size

Graph.**size**(*weighted=False*)

Return the number of edges.

Parameters **weighted** : boolean, optional (default=False)

If True return the sum of the edge weights.

Returns **nedges** : int

The number of edges in the graph.

See Also:

`number_of_edges`

Examples

```
>>> G = nx.Graph()   # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> 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(weighted=True)
6.0
```

networkx.Graph.number_of_edges

`Graph.`**`number_of_edges`**(*u=None*, *v=None*)

Return 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.

See Also:

`size`

Examples

```
>>> G = nx.Graph()   # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.number_of_edges()
3
>>> G.number_of_edges(0,1)
1
>>> e = (0,1)
>>> G.number_of_edges(*e)
1
```

networkx.Graph.nodes_with_selfloops

Graph.**nodes_with_selfloops**()

Return a list of nodes with self loops.

A node with a self loop has an edge with both ends adjacent to that node.

Returns **nodelist** : list

A list of 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)
>>> G.nodes_with_selfloops()
[1]
```

networkx.Graph.selfloop_edges

Graph.**selfloop_edges**(*data=False*)

Return a list of selfloop edges.

A selfloop edge has the same node at both ends.

Parameters **data** : bool, optional (default=False)

Return selfloop edges as two tuples (u,v) (data=False) or three-tuples (u,v,data) (data=True)

Returns **edgelist** : list of edge tuples

A list of all selfloop edges.

See Also:

selfloop_nodes, number_of_selfloops

Examples

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge(1,1)
>>> G.add_edge(1,2)
>>> G.selfloop_edges()
[(1, 1)]
>>> G.selfloop_edges(data=True)
[(1, 1, {})]
```

networkx.Graph.number_of_selfloops

Graph.**number_of_selfloops**()
Return the number of selfloop edges.

A selfloop edge has the same node at both ends.

Returns **nloops** : int

The number of selfloops.

See Also:

selfloop_nodes, selfloop_edges

Examples

```
>>> G=nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge(1,1)
>>> G.add_edge(1,2)
>>> G.number_of_selfloops()
1
```

Making copies and subgraphs

Graph.copy()	Return a copy of the graph.
Graph.to_undirected()	Return an undirected copy of the graph.
Graph.to_directed()	Return a directed representation of the graph.
Graph.subgraph(nbunch)	Return the subgraph induced on nodes in nbunch.

networkx.Graph.copy

Graph.**copy**()
Return a copy of the graph.

Returns **G** : Graph

A copy of the graph.

See Also:

to_directed return a directed copy of the graph.

Notes

This makes a complete copy of the graph including all of the node or edge attributes.

Examples

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> H = G.copy()
```

networkx.Graph.to_undirected

Graph.**to_undirected**()

Return 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=DiGraph(D) which returns a shallow copy of the data.

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

Examples

```
>>> G = nx.Graph()   # or MultiGraph, etc
>>> G.add_path([0,1])
>>> H = G.to_directed()
>>> H.edges()
[(0, 1), (1, 0)]
>>> G2 = H.to_undirected()
>>> G2.edges()
[(0, 1)]
```

networkx.Graph.to_directed

Graph.**to_directed**()

Return 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, http://docs.python.org/library/copy.html.

Examples

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

If already directed, return a (deep) copy

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

networkx.Graph.subgraph

Graph.**subgraph**(*nbunch*)

Return the subgraph induced on nodes in nbunch.

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

Parameters **nbunch** : list, iterable

A container of nodes which will be iterated through once.

Returns **G** : Graph

A subgraph of the graph with the same edge attributes.

Notes

The graph, edge or node attributes just point to the original graph. So changes to the node or edge structure will not be reflected in the original graph while changes to the attributes will.

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

If edge attributes are containers, a deep copy can be obtained using: G.subgraph(nbunch).copy()

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)])

Examples

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> H = G.subgraph([0,1,2])
>>> H.edges()
[(0, 1), (1, 2)]
```

3.2.2 DiGraph - Directed graphs with self loops

Overview

networkx.**DiGraph**(*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.

Edges are represented as links between nodes with optional key/value attributes.

Parameters **data** : input graph

Data to initialize graph. If 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 NumPy matrix or 2d ndarray, a SciPy sparse matrix, or a 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.Graph()
>>> H.add_path([0,1,2,3,4,5,6,7,8,9])
>>> 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.node

```
>>> G.add_node(1, time='5pm')
>>> G.add_nodes_from([3], time='2pm')
>>> G.node[1]
{'time': '5pm'}
>>> G.node[1]['room'] = 714
>>> G.nodes(data=True)
[(1, {'room': 714, 'time': '5pm'}), (3, {'time': '2pm'})]
```

Warning: adding a node to G.node does not add it to the graph.

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

```
>>> 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.edge[1][2]['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 keyed by neighbor to edge attributes
...            # Note: you should not change this dict manually!
{2: {'color': 'blue', 'weight': 4}}
```

The fastest way to traverse all edges of a graph is via adjacency_iter(), but the edges() method is often more convenient.

```
>>> for n,nbrsdict in G.adjacency_iter():
...     for nbr,eattr in nbrsdict.items():
...        if 'weight' in eattr:
...            (n,nbr,eattr['weight'])
(1, 2, 4)
(2, 3, 8)
>>> [ (u,v,edata['weight']) for u,v,edata in G.edges(data=True) if 'weight' in edata ]
[(1, 2, 4), (2, 3, 8)]
```

Reporting:

Simple graph information is obtained using methods. Iterator versions of many reporting methods exist for efficiency. Methods exist for reporting nodes(), edges(), neighbors() and degree() as well as the number of nodes and edges.

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

Adding and removing nodes and edges

DiGraph.__init__(**attr[, data])	Initialize a graph with edges, name, graph attributes.
DiGraph.add_node(n, **attr[, attr_dict])	Add a single node n and update node attributes.
DiGraph.add_nodes_from(nodes, **attr)	Add multiple nodes.
DiGraph.remove_node(n)	Remove node n.
DiGraph.remove_nodes_from(nbunch)	Remove multiple nodes.
DiGraph.add_edge(u, v, **attr[, attr_dict])	Add an edge between u and v.
DiGraph.add_edges_from(ebunch, **attr[, ...])	Add all the edges in ebunch.
DiGraph.add_weighted_edges_from(ebunch, **attr)	Add all the edges in ebunch as weighted edges with specified weights.
DiGraph.remove_edge(u, v)	Remove the edge between u and v.
DiGraph.remove_edges_from(ebunch)	Remove all edges specified in ebunch.
DiGraph.add_star(nodes, **attr)	Add a star.
DiGraph.add_path(nodes, **attr)	Add a path.
DiGraph.add_cycle(nodes, **attr)	Add a cycle.
DiGraph.clear()	Remove all nodes and edges from the graph.

networkx.DiGraph.__init__

DiGraph.**__init__**(*data=None*, ***attr*)

Initialize a graph with edges, name, graph attributes.

Parameters **data** : input graph

Data to initialize graph. If 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 NumPy matrix or 2d ndarray, a SciPy sparse matrix, or a PyGraphviz graph.

name : string, optional (default='')

An optional name for the 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'}
```

networkx.DiGraph.add_node

DiGraph.**add_node**(*n*, *attr_dict=None*, ***attr*)

Add a single node n and update node attributes.

Parameters **n** : node

A node can be any hashable Python object except None.

attr_dict : dictionary, optional (default= no attributes)

Dictionary of node attributes. Key/value pairs will update existing data associated with the node.

attr : keyword arguments, optional

Set or change 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))
```

networkx.DiGraph.add_nodes_from

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

Add multiple nodes.

Parameters **nodes** : 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 generally.

See Also:

add_node

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.node[1]['size']
11
>>> H = nx.Graph()
>>> H.add_nodes_from(G.nodes(data=True))
>>> H.node[1]['size']
11
```

networkx.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.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> G.edges()
[(0, 1), (1, 2)]
>>> G.remove_node(1)
>>> G.edges()
[]
```

networkx.DiGraph.remove_nodes_from

DiGraph.**remove_nodes_from**(*nbunch*)

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`

Examples

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

networkx.DiGraph.add_edge

DiGraph.**add_edge**(*u*, *v*, *attr_dict=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 providing a dictionary with key/value pairs. See examples below.

Parameters **u,v** : nodes

Nodes can be, for example, strings or numbers. Nodes must be hashable (and not None) Python objects.

attr_dict : dictionary, optional (default= no attributes)

Dictionary of edge attributes. Key/value pairs will update existing data associated with the edge.

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.

NetworkX algorithms designed for weighted graphs use as the edge weight a numerical value assigned to the keyword 'weight'.

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)
```

networkx.DiGraph.add_edges_from

DiGraph.**add_edges_from**(*ebunch*, *attr_dict=None*, ***attr*)

Add all the edges in ebunch.

Parameters **ebunch** : container of edges

Each edge given in the container will be added to the graph. The edges must be given as as 2-tuples (u,v) or 3-tuples (u,v,d) where d is a dictionary containing edge data.

attr_dict : dictionary, optional (default= no attributes)

Dictionary of edge attributes. Key/value pairs will update existing data associated with each edge.

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.

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')
```

networkx.DiGraph.add_weighted_edges_from

DiGraph.**add_weighted_edges_from**(*ebunch*, ***attr*)

Add all the edges in ebunch as weighted edges with specified weights.

Parameters **ebunch** : 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.

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 has no effect but any edge data will be updated when each duplicate edge is added.

Examples

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

networkx.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
>>> G.add_path([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
```

networkx.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.Graph()   # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> ebunch=[(1,2),(2,3)]
>>> G.remove_edges_from(ebunch)
```

networkx.DiGraph.add_star

DiGraph.**add_star**(*nodes*, ***attr*)

Add a star.

The first node in nodes is the middle of the star. It is connected to all other nodes.

Parameters **nodes** : 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()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_star([0,1,2,3])
>>> G.add_star([10,11,12],weight=2)
```

networkx.DiGraph.add_path

DiGraph.**add_path**(*nodes*, ***attr*)

Add a path.

Parameters **nodes** : 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()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.add_path([10,11,12],weight=7)
```

networkx.DiGraph.add_cycle

DiGraph.**add_cycle**(*nodes*, ***attr*)

Add a cycle.

Parameters **nodes: 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
>>> G.add_cycle([0,1,2,3])
>>> G.add_cycle([10,11,12],weight=7)
```

networkx.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.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.clear()
>>> G.nodes()
[]
>>> G.edges()
[]
```

Iterating over nodes and edges

DiGraph.nodes([data])	Return a list of the nodes in the graph.
DiGraph.nodes_iter([data])	Return an iterator over the nodes.
DiGraph.__iter__()	Iterate over the nodes.
DiGraph.edges([nbunch, data])	Return a list of edges.
DiGraph.edges_iter([nbunch, data])	Return an iterator over the edges.
DiGraph.out_edges([nbunch, data])	Return a list of edges.
DiGraph.out_edges_iter([nbunch, data])	Return an iterator over the edges.
DiGraph.in_edges([nbunch, data])	Return a list of the incoming edges.
DiGraph.in_edges_iter([nbunch, data])	Return an iterator over the incoming edges.
DiGraph.get_edge_data(u, v[, default])	Return the attribute dictionary associated with edge (u,v).
DiGraph.neighbors(n)	Return a list of successor nodes of n.
DiGraph.neighbors_iter(n)	Return an iterator over successor nodes of n.
DiGraph.__getitem__(n)	Return a dict of neighbors of node n.
DiGraph.successors(n)	Return a list of successor nodes of n.
DiGraph.successors_iter(n)	Return an iterator over successor nodes of n.
DiGraph.predecessors(n)	Return a list of predecessor nodes of n.
DiGraph.predecessors_iter(n)	Return an iterator over predecessor nodes of n.
DiGraph.adjacency_list()	Return an adjacency list representation of the graph.
DiGraph.adjacency_iter()	Return an iterator of (node, adjacency dict) tuples for all nodes.
DiGraph.nbunch_iter([nbunch])	Return an iterator of nodes contained in nbunch that are also in the graph.

networkx.DiGraph.nodes

DiGraph.**nodes**(*data=False*)

Return a list of the nodes in the graph.

Parameters **data** : boolean, optional (default=False)

If False return a list of nodes. If True return a two-tuple of node and node data dictionary

Returns **nlist** : list

A list of nodes. If data=True a list of two-tuples containing (node, node data dictionary).

Examples

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> G.nodes()
[0, 1, 2]
>>> G.add_node(1, time='5pm')
>>> G.nodes(data=True)
[(0, {}), (1, {'time': '5pm'}), (2, {})]
```

networkx.DiGraph.nodes_iter

DiGraph.**nodes_iter**(*data=False*)

Return an iterator over the nodes.

Parameters **data** : boolean, optional (default=False)

If False the iterator returns nodes. If True return a two-tuple of node and node data dictionary

Returns **niter** : iterator

An iterator over nodes. If data=True the iterator gives two-tuples containing (node, node data, dictionary)

Notes

If the node data is not required it is simpler and equivalent to use the expression 'for n in G'.

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

Examples

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

>>> [d for n,d in G.nodes_iter(data=True)]
[{}, {}, {}]
```

networkx.DiGraph.__iter__

DiGraph.**__iter__**()

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

Returns **niter** : iterator

An iterator over all nodes in the graph.

Examples

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

networkx.DiGraph.edges

DiGraph.**edges**(*nbunch=None*, *data=False*)

Return a list of edges.

Edges are returned as tuples with optional data in the order (node, neighbor, data).

Parameters **nbunch** : iterable container, optional (default= all nodes)

A container of nodes. The container will be iterated through once.

data : bool, optional (default=False)

Return two tuples (u,v) (False) or three-tuples (u,v,data) (True).

Returns **edge_list: list of edge tuples** :

Edges that are adjacent to any node in nbunch, or a list of all edges if nbunch is not specified.

See Also:

`edges_iter` return an iterator over the edges

Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored.

Examples

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.edges()
[(0, 1), (1, 2), (2, 3)]
>>> G.edges(data=True) # default edge data is {} (empty dictionary)
[(0, 1, {}), (1, 2, {}), (2, 3, {})]
>>> G.edges([0,3])
[(0, 1), (3, 2)]
>>> G.edges(0)
[(0, 1)]
```

networkx.DiGraph.edges_iter

DiGraph.**edges_iter**(*nbunch=None*, *data=False*)

Return an iterator over the edges.

Edges are returned as tuples with optional data in the order (node, neighbor, data).

Parameters **nbunch** : iterable container, optional (default= all nodes)

A container of nodes. The container will be iterated through once.

data : bool, optional (default=False)

If True, return edge attribute dict in 3-tuple (u,v,data).

Returns **edge_iter** : iterator

An iterator of (u,v) or (u,v,d) tuples of edges.

See Also:

`edges` return a list of edges

Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored.

Examples

```
>>> G = nx.DiGraph()    # or MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> [e for e in G.edges_iter()]
[(0, 1), (1, 2), (2, 3)]
>>> list(G.edges_iter(data=True)) # default data is {} (empty dict)
[(0, 1, {}), (1, 2, {}), (2, 3, {})]
>>> list(G.edges_iter([0,2]))
[(0, 1), (2, 3)]
>>> list(G.edges_iter(0))
[(0, 1)]
```

networkx.DiGraph.out_edges

DiGraph.**out_edges**(*nbunch=None*, *data=False*)

Return a list of edges.

Edges are returned as tuples with optional data in the order (node, neighbor, data).

Parameters **nbunch** : iterable container, optional (default= all nodes)

A container of nodes. The container will be iterated through once.

data : bool, optional (default=False)

Return two tuples (u,v) (False) or three-tuples (u,v,data) (True).

Returns **edge_list: list of edge tuples** :

Edges that are adjacent to any node in nbunch, or a list of all edges if nbunch is not specified.

See Also:

edges_iter return an iterator over the edges

Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored.

Examples

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.edges()
[(0, 1), (1, 2), (2, 3)]
>>> G.edges(data=True) # default edge data is {} (empty dictionary)
[(0, 1, {}), (1, 2, {}), (2, 3, {})]
>>> G.edges([0,3])
[(0, 1), (3, 2)]
```

```
>>> G.edges(0)
[(0, 1)]
```

networkx.DiGraph.out_edges_iter

DiGraph.**out_edges_iter**(*nbunch=None*, *data=False*)

Return an iterator over the edges.

Edges are returned as tuples with optional data in the order (node, neighbor, data).

Parameters **nbunch** : iterable container, optional (default= all nodes)

A container of nodes. The container will be iterated through once.

data : bool, optional (default=False)

If True, return edge attribute dict in 3-tuple (u,v,data).

Returns **edge_iter** : iterator

An iterator of (u,v) or (u,v,d) tuples of edges.

See Also:

edges return a list of edges

Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored.

Examples

```
>>> G = nx.DiGraph()   # or MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> [e for e in G.edges_iter()]
[(0, 1), (1, 2), (2, 3)]
>>> list(G.edges_iter(data=True)) # default data is {} (empty dict)
[(0, 1, {}), (1, 2, {}), (2, 3, {})]
>>> list(G.edges_iter([0,2]))
[(0, 1), (2, 3)]
>>> list(G.edges_iter(0))
[(0, 1)]
```

networkx.DiGraph.in_edges

DiGraph.**in_edges**(*nbunch=None*, *data=False*)

Return a list of the incoming edges.

See Also:

edges return a list of edges

networkx.DiGraph.in_edges_iter

DiGraph.**in_edges_iter**(*nbunch=None*, *data=False*)

Return an iterator over the incoming edges.

Parameters **nbunch** : iterable container, optional (default= all nodes)

A container of nodes. The container will be iterated through once.

data : bool, optional (default=False)

If True, return edge attribute dict in 3-tuple (u,v,data).

Returns **in_edge_iter** : iterator

An iterator of (u,v) or (u,v,d) tuples of incoming edges.

See Also:

edges_iter return an iterator of edges

networkx.DiGraph.get_edge_data

DiGraph.**get_edge_data**(*u*, *v*, *default=None*)

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

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.

Notes

It is faster to use G[u][v].

```
>>> G = nx.Graph()   # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G[0][1]
{}
```

Warning: Assigning G[u][v] corrupts the graph data structure. But it is safe to assign attributes to that dictionary,

```
>>> G[0][1]['weight'] = 7
>>> G[0][1]['weight']
7
>>> G[1][0]['weight']
7
```

Examples

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

networkx.DiGraph.neighbors

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

Return a list of successor nodes of n.

neighbors() and successors() are the same function.

networkx.DiGraph.neighbors_iter

DiGraph.**neighbors_iter**(*n*)

Return an iterator over successor nodes of n.

neighbors_iter() and successors_iter() are the same.

networkx.DiGraph.__getitem__

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

Return a dict of neighbors of node n. Use the expression '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 similar to G.neighbors(n) but the internal data dictionary is returned instead of a list.

Assigning G[n] will corrupt the internal graph data structure. Use G[n] for reading data only.

Examples

```
>>> G = nx.Graph()   # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G[0]
{1: {}}
```

networkx.DiGraph.successors

DiGraph.**successors**(*n*)
Return a list of successor nodes of n.

neighbors() and successors() are the same function.

networkx.DiGraph.successors_iter

DiGraph.**successors_iter**(*n*)
Return an iterator over successor nodes of n.

neighbors_iter() and successors_iter() are the same.

networkx.DiGraph.predecessors

DiGraph.**predecessors**(*n*)
Return a list of predecessor nodes of n.

networkx.DiGraph.predecessors_iter

DiGraph.**predecessors_iter**(*n*)
Return an iterator over predecessor nodes of n.

networkx.DiGraph.adjacency_list

DiGraph.**adjacency_list**()
Return an adjacency list representation of the graph.

The output adjacency list is in the order of G.nodes(). For directed graphs, only outgoing adjacencies are included.

Returns **adj_list** : lists of lists

The adjacency structure of the graph as a list of lists.

See Also:

adjacency_iter

Examples

```
>>> G = nx.Graph()   # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.adjacency_list() # in order given by G.nodes()
[[1], [0, 2], [1, 3], [2]]
```

networkx.DiGraph.adjacency_iter

DiGraph.**adjacency_iter**()
Return an iterator of (node, adjacency dict) tuples for all nodes.

This is the fastest way to look at every edge. For directed graphs, only outgoing adjacencies are included.

Returns **adj_iter** : iterator

An iterator of (node, adjacency dictionary) for all nodes in the graph.

See Also:

`adjacency_list`

Examples

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

networkx.DiGraph.nbunch_iter

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

Return an iterator of 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** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

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 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.

Information about graph structure

DiGraph.has_node(n)	Return True if the graph contains the node n.
DiGraph.__contains__(n)	Return True if n is a node, False otherwise. Use the expression
DiGraph.has_edge(u, v)	Return True if the edge (u,v) is in the graph.
DiGraph.order()	Return the number of nodes in the graph.
DiGraph.number_of_nodes()	Return the number of nodes in the graph.
DiGraph.__len__()	Return the number of nodes.
DiGraph.degree([nbunch, weighted])	Return the degree of a node or nodes.
DiGraph.degree_iter([nbunch, weighted])	Return an iterator for (node, degree).
DiGraph.in_degree([nbunch, weighted])	Return the in-degree of a node or nodes.
DiGraph.in_degree_iter([nbunch, weighted])	Return an iterator for (node, in-degree).
DiGraph.out_degree([nbunch, weighted])	Return the out-degree of a node or nodes.
DiGraph.out_degree_iter([nbunch, weighted])	Return an iterator for (node, out-degree).
DiGraph.size([weighted])	Return the number of edges.
DiGraph.number_of_edges([u, v])	Return the number of edges between two nodes.
DiGraph.nodes_with_selfloops()	Return a list of nodes with self loops.
DiGraph.selfloop_edges([data])	Return a list of selfloop edges.
DiGraph.number_of_selfloops()	Return the number of selfloop edges.

networkx.DiGraph.has_node

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

Return True if the graph contains the node n.

Parameters **n** : node

Examples

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> G.has_node(0)
True
```

It is more readable and simpler to use

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

networkx.DiGraph.__contains__

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

Return True if n is a node, False otherwise. Use the expression 'n in G'.

Examples

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> 1 in G
True
```

networkx.DiGraph.has_edge

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

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

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

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

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

The following syntax are all equivalent:

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

networkx.DiGraph.order

DiGraph.**order**()

Return the number of nodes in the graph.

Returns **nnodes** : int

The number of nodes in the graph.

See Also:

number_of_nodes, __len__

networkx.DiGraph.number_of_nodes

DiGraph.**number_of_nodes**()

Return the number of nodes in the graph.

Returns **nnodes** : int

The number of nodes in the graph.

See Also:

order, __len__

Examples

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

networkx.DiGraph.__len__

DiGraph.**__len__**()

Return the number of nodes. Use the expression 'len(G)'.

Returns **nnodes** : int

The number of nodes in the graph.

Examples

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> len(G)
4
```

networkx.DiGraph.degree

DiGraph.**degree**(*nbunch=None*, *weighted=False*)

Return the degree of a node or nodes.

The node degree is the number of edges adjacent to that node.

Parameters **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

weighted : bool, optional (default=False)

If True return the sum of edge weights adjacent to the node.

Returns **nd** : dictionary, or number

A dictionary with nodes as keys and degree as values or a number if a single node is specified.

Examples

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

networkx.DiGraph.degree_iter

DiGraph.**degree_iter**(*nbunch=None*, *weighted=False*)

Return an iterator for (node, degree).

The node degree is the number of edges adjacent to the node.

Parameters **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

weighted : bool, optional (default=False)

If True return the sum of edge weights adjacent to the node.

Returns **nd_iter** : an iterator

The iterator returns two-tuples of (node, degree).

See Also:

degree, in_degree, out_degree, in_degree_iter, out_degree_iter

Examples

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

networkx.DiGraph.in_degree

DiGraph.**in_degree**(*nbunch=None*, *weighted=False*)

Return the in-degree of a node or nodes.

The node in-degree is the number of edges pointing in to the node.

Parameters **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

weighted : bool, optional (default=False)

If True return the sum of edge weights adjacent to the node.

Returns **nd** : dictionary, or number

A dictionary with nodes as keys and in-degree as values or a number if a single node is specified.

See Also:

degree, out_degree, in_degree_iter

Examples

```
>>> G = nx.DiGraph()    # or MultiDiGraph
>>> G.add_path([0,1,2,3])
>>> G.in_degree(0)
0
>>> G.in_degree([0,1])
{0: 0, 1: 1}
>>> list(G.in_degree([0,1]).values())
[0, 1]
```

networkx.DiGraph.in_degree_iter

DiGraph.**in_degree_iter**(*nbunch=None*, *weighted=False*)

Return an iterator for (node, in-degree).

The node in-degree is the number of edges pointing in to the node.

Parameters **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

weighted : bool, optional (default=False)

If True return the sum of edge weights adjacent to the node.

Returns **nd_iter** : an iterator

The iterator returns two-tuples of (node, in-degree).

See Also:

degree, in_degree, out_degree, out_degree_iter

Examples

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

networkx.DiGraph.out_degree

DiGraph.**out_degree**(*nbunch=None*, *weighted=False*)

Return the out-degree of a node or nodes.

The node out-degree is the number of edges pointing out of the node.

Parameters **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

weighted : bool, optional (default=False)

If True return the sum of edge weights adjacent to the node.

Returns **nd** : dictionary, or number

A dictionary with nodes as keys and out-degree as values or a number if a single node is specified.

Examples

```
>>> G = nx.DiGraph()    # or MultiDiGraph
>>> G.add_path([0,1,2,3])
>>> G.out_degree(0)
1
>>> G.out_degree([0,1])
{0: 1, 1: 1}
>>> list(G.out_degree([0,1]).values())
[1, 1]
```

networkx.DiGraph.out_degree_iter

DiGraph.**out_degree_iter**(*nbunch=None*, *weighted=False*)

Return an iterator for (node, out-degree).

The node out-degree is the number of edges pointing out of the node.

Parameters **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

weighted : bool, optional (default=False)

If True return the sum of edge weights adjacent to the node.

Returns **nd_iter** : an iterator

The iterator returns two-tuples of (node, out-degree).

See Also:

degree, in_degree, out_degree, in_degree_iter

Examples

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

networkx.DiGraph.size

DiGraph.**size**(*weighted=False*)

Return the number of edges.

Parameters **weighted** : boolean, optional (default=False)

If True return the sum of the edge weights.

Returns **nedges** : int

The number of edges in the graph.

See Also:

number_of_edges

Examples

```
>>> G = nx.Graph()   # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> 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(weighted=True)
6.0
```

networkx.DiGraph.number_of_edges

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

Return 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.

See Also:

size

Examples

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

```
1
>>> e = (0,1)
>>> G.number_of_edges(*e)
1
```

networkx.DiGraph.nodes_with_selfloops

DiGraph.**nodes_with_selfloops**()

Return a list of nodes with self loops.

A node with a self loop has an edge with both ends adjacent to that node.

Returns **nodelist** : list

A list of 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)
>>> G.nodes_with_selfloops()
[1]
```

networkx.DiGraph.selfloop_edges

DiGraph.**selfloop_edges**(*data=False*)

Return a list of selfloop edges.

A selfloop edge has the same node at both ends.

Parameters **data** : bool, optional (default=False)

Return selfloop edges as two tuples (u,v) (data=False) or three-tuples (u,v,data) (data=True)

Returns **edgelist** : list of edge tuples

A list of all selfloop edges.

See Also:

selfloop_nodes, number_of_selfloops

Examples

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge(1,1)
>>> G.add_edge(1,2)
>>> G.selfloop_edges()
[(1, 1)]
```

```
>>> G.selfloop_edges(data=True)
[(1, 1, {})]
```

networkx.DiGraph.number_of_selfloops

DiGraph.**number_of_selfloops**()

Return the number of selfloop edges.

A selfloop edge has the same node at both ends.

Returns **nloops** : int

The number of selfloops.

See Also:

selfloop_nodes, selfloop_edges

Examples

```
>>> G=nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge(1,1)
>>> G.add_edge(1,2)
>>> G.number_of_selfloops()
1
```

Making copies and subgraphs

DiGraph.copy()	Return a copy of the graph.
DiGraph.to_undirected([reciprocal])	Return an undirected representation of the digraph.
DiGraph.to_directed()	Return a directed copy of the graph.
DiGraph.subgraph(nbunch)	Return the subgraph induced on nodes in nbunch.
DiGraph.reverse([copy])	Return the reverse of the graph.

networkx.DiGraph.copy

DiGraph.**copy**()

Return a copy of the graph.

Returns **G** : Graph

A copy of the graph.

See Also:

to_directed return a directed copy of the graph.

Notes

This makes a complete copy of the graph including all of the node or edge attributes.

Examples

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> H = G.copy()
```

networkx.DiGraph.to_undirected

DiGraph.**to_undirected**(*reciprocal=False*)

Return an undirected representation of the digraph.

Parameters **reciprocal** : bool (optional)

If True only keep edges that appear in both directions in the original digraph.

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.

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, http://docs.python.org/library/copy.html.

networkx.DiGraph.to_directed

DiGraph.**to_directed**()

Return a directed copy of the graph.

Returns **G** : DiGraph

A deepcopy of the graph.

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, http://docs.python.org/library/copy.html.

Examples

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

If already directed, return a (deep) copy

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

networkx.DiGraph.subgraph

DiGraph.**subgraph**(*nbunch*)

Return the subgraph induced on nodes in nbunch.

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

Parameters **nbunch** : list, iterable

A container of nodes which will be iterated through once.

Returns **G** : Graph

A subgraph of the graph with the same edge attributes.

Notes

The graph, edge or node attributes just point to the original graph. So changes to the node or edge structure will not be reflected in the original graph while changes to the attributes will.

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

If edge attributes are containers, a deep copy can be obtained using: G.subgraph(nbunch).copy()

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)])

Examples

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> H = G.subgraph([0,1,2])
>>> H.edges()
[(0, 1), (1, 2)]
```

networkx.DiGraph.reverse

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

Return 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, reverse the reverse graph is created using the original graph (this changes the original graph).

3.2.3 MultiGraph - Undirected graphs with self loops and parallel edges

Overview

networkx.**MultiGraph**(*data=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.

Edges are represented as links between nodes with optional key/value attributes.

Parameters **data** : input graph

Data to initialize graph. If 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 NumPy matrix or 2d ndarray, a SciPy sparse matrix, or a PyGraphviz graph.

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.Graph()
>>> H.add_path([0,1,2,3,4,5,6,7,8,9])
>>> 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. 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.

```
>>> G.add_edges_from([(4,5,dict(route=282)), (4,5,dict(route=37))])
>>> G[4]
{3: {0: {}}, 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.MultiGraph(day="Friday")
>>> G.graph
{'day': 'Friday'}
```

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

```
>>> G.add_node(1, time='5pm')
>>> G.add_nodes_from([3], time='2pm')
>>> G.node[1]
{'time': '5pm'}
>>> G.node[1]['room'] = 714
>>> G.nodes(data=True)
[(1, {'room': 714, 'time': '5pm'}), (3, {'time': '2pm'})]
```

Warning: adding a node to G.node does not add it to the graph.

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

```
>>> 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][0]['weight'] = 4.7
>>> G.edge[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 keyed by neighbor to edge attributes
...            # Note: you should not change this dict manually!
{2: {0: {'weight': 4}, 1: {'color': 'blue'}}}
```

The fastest way to traverse all edges of a graph is via adjacency_iter(), but the edges() method is often more convenient.

```
>>> for n,nbrsdict in G.adjacency_iter():
...     for nbr,keydict in nbrsdict.items():
...        for key,eattr in keydict.items():
...            if 'weight' in eattr:
...                (n,nbr,eattr['weight'])
(1, 2, 4)
(2, 1, 4)
(2, 3, 8)
(3, 2, 8)
>>> [ (u,v,edata['weight']) for u,v,edata in G.edges(data=True) if 'weight' in edata ]
[(1, 2, 4), (2, 3, 8)]
```

Reporting:

Simple graph information is obtained using methods. Iterator versions of many reporting methods exist for efficiency. Methods exist for reporting nodes(), edges(), neighbors() and degree() as well as the number of nodes and edges.

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

Adding and removing nodes and edges

MultiGraph.__init__(**attr[, data])	Initialize a graph with edges, name, graph attributes.
MultiGraph.add_node(n, **attr[, attr_dict])	Add a single node n and update node attributes.
MultiGraph.add_nodes_from(nodes, **attr)	Add multiple nodes.
MultiGraph.remove_node(n)	Remove node n.
MultiGraph.remove_nodes_from(nodes)	Remove multiple nodes.
MultiGraph.add_edge(u, v, **attr[, key, ...])	Add an edge between u and v.
MultiGraph.add_edges_from(ebunch, **attr[, ...])	Add all the edges in ebunch.
MultiGraph.add_weighted_edges_from(ebunch, ...)	Add all the edges in ebunch as weighted edges with specified weights.
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.add_star(nodes, **attr)	Add a star.
MultiGraph.add_path(nodes, **attr)	Add a path.
MultiGraph.add_cycle(nodes, **attr)	Add a cycle.
MultiGraph.clear()	Remove all nodes and edges from the graph.

networkx.MultiGraph.__init__

MultiGraph.**__init__**(*data=None*, ***attr*)

Initialize a graph with edges, name, graph attributes.

Parameters **data** : input graph

Data to initialize graph. If 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 NumPy matrix or 2d ndarray, a SciPy sparse matrix, or a PyGraphviz graph.

name : string, optional (default='')

An optional name for the 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'}
```

networkx.MultiGraph.add_node

MultiGraph.**add_node**(*n*, *attr_dict=None*, ***attr*)

Add a single node n and update node attributes.

Parameters **n** : node

A node can be any hashable Python object except None.

attr_dict : dictionary, optional (default= no attributes)

Dictionary of node attributes. Key/value pairs will update existing data associated with the node.

attr : keyword arguments, optional

Set or change 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))
```

networkx.MultiGraph.add_nodes_from

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

Add multiple nodes.

Parameters **nodes** : 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 generally.

See Also:

add_node

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.node[1]['size']
11
>>> H = nx.Graph()
>>> H.add_nodes_from(G.nodes(data=True))
>>> H.node[1]['size']
11
```

networkx.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.Graph()   # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> G.edges()
[(0, 1), (1, 2)]
>>> G.remove_node(1)
>>> G.edges()
[]
```

networkx.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

Examples

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

networkx.MultiGraph.add_edge

MultiGraph.**add_edge**(*u*, *v*, *key=None*, *attr_dict=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 providing a dictionary with key/value pairs. See examples below.

Parameters **u,v** : 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_dict : dictionary, optional (default= no attributes)

Dictionary of edge attributes. Key/value pairs will update existing data associated with the edge.

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

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.

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, 2, key=0, weight=4)    # update data for key=0
>>> G.add_edge(1, 3, weight=7, capacity=15, length=342.7)
```

networkx.MultiGraph.add_edges_from

MultiGraph.**add_edges_from**(*ebunch*, *attr_dict=None*, ***attr*)

Add all the edges in ebunch.

Parameters **ebunch** : 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 attribute dict d, or
- 4-tuples (u,v,k,d) for an edge identified by key k

attr_dict : dictionary, optional (default= no attributes)

Dictionary of edge attributes. Key/value pairs will update existing data associated with each edge.

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.

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')
```

networkx.MultiGraph.add_weighted_edges_from

MultiGraph.**add_weighted_edges_from**(*ebunch*, ***attr*)

Add all the edges in ebunch as weighted edges with specified weights.

Parameters **ebunch** : 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.

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 has no effect but any edge data will be updated when each duplicate edge is added.

Examples

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

networkx.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 (abritrary) edge between u and v.

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()
>>> G.add_path([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)])
>>> G.remove_edge(1,2) # remove a single (arbitrary) edge
```

For edges with keys

```
>>> G = nx.MultiGraph()   # or MultiDiGraph, etc
>>> G.add_edge(1,2,key='first')
>>> G.add_edge(1,2,key='second')
>>> G.remove_edge(1,2,key='second')
```

networkx.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) All edges between u and v are removed.
- 3-tuples (u,v,key) The edge identified by key is removed.

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.MultiGraph() # or MultiDiGraph
>>> G.add_path([0,1,2,3])
>>> ebunch=[(1,2),(2,3)]
>>> G.remove_edges_from(ebunch)
```

Removing multiple copies of edges

```
>>> G = nx.MultiGraph()
>>> G.add_edges_from([(1,2),(1,2),(1,2)])
>>> G.remove_edges_from([(1,2),(1,2)])
>>> G.edges()
[(1, 2)]
>>> G.remove_edges_from([(1,2),(1,2)]) # silently ignore extra copy
```

```
>>> G.edges() # now empty graph
[]
```

networkx.MultiGraph.add_star

MultiGraph.**add_star**(*nodes*, ***attr*)

Add a star.

The first node in nodes is the middle of the star. It is connected to all other nodes.

Parameters **nodes** : 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()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_star([0,1,2,3])
>>> G.add_star([10,11,12],weight=2)
```

networkx.MultiGraph.add_path

MultiGraph.**add_path**(*nodes*, ***attr*)

Add a path.

Parameters **nodes** : 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()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.add_path([10,11,12],weight=7)
```

networkx.MultiGraph.add_cycle

MultiGraph.**add_cycle**(*nodes*, ***attr*)

Add a cycle.

Parameters **nodes: 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
>>> G.add_cycle([0,1,2,3])
>>> G.add_cycle([10,11,12],weight=7)
```

networkx.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.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.clear()
>>> G.nodes()
[]
>>> G.edges()
[]
```

Iterating over nodes and edges

MultiGraph.nodes([data])	Return a list of the nodes in the graph.
MultiGraph.nodes_iter([data])	Return an iterator over the nodes.
MultiGraph.__iter__()	Iterate over the nodes.
MultiGraph.edges([nbunch, data, keys])	Return a list of edges.
MultiGraph.edges_iter([nbunch, data, keys])	Return an iterator over the edges.
MultiGraph.get_edge_data(u, v[, key, default])	Return the attribute dictionary associated with edge (u,v).
MultiGraph.neighbors(n)	Return a list of the nodes connected to the node n.
MultiGraph.neighbors_iter(n)	Return an iterator over all neighbors of node n.
MultiGraph.__getitem__(n)	Return a dict of neighbors of node n.
MultiGraph.adjacency_list()	Return an adjacency list representation of the graph.
MultiGraph.adjacency_iter()	Return an iterator of (node, adjacency dict) tuples for all nodes.
MultiGraph.nbunch_iter([nbunch])	Return an iterator of nodes contained in nbunch that are also in the graph.

networkx.MultiGraph.nodes

MultiGraph.**nodes**(*data=False*)

Return a list of the nodes in the graph.

Parameters **data** : boolean, optional (default=False)

If False return a list of nodes. If True return a two-tuple of node and node data dictionary

Returns **nlist** : list

A list of nodes. If data=True a list of two-tuples containing (node, node data dictionary).

Examples

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> G.nodes()
[0, 1, 2]
>>> G.add_node(1, time='5pm')
>>> G.nodes(data=True)
[(0, {}), (1, {'time': '5pm'}), (2, {})]
```

networkx.MultiGraph.nodes_iter

MultiGraph.**nodes_iter**(*data=False*)

Return an iterator over the nodes.

Parameters **data** : boolean, optional (default=False)

If False the iterator returns nodes. If True return a two-tuple of node and node data dictionary

Returns **niter** : iterator

An iterator over nodes. If data=True the iterator gives two-tuples containing (node, node data, dictionary)

Notes

If the node data is not required it is simpler and equivalent to use the expression 'for n in G'.

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

Examples

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

>>> [d for n,d in G.nodes_iter(data=True)]
[{}, {}, {}]
```

networkx.MultiGraph.__iter__

MultiGraph.**__iter__**()

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

Returns **niter** : iterator

An iterator over all nodes in the graph.

Examples

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

networkx.MultiGraph.edges

MultiGraph.**edges**(*nbunch=None*, *data=False*, *keys=False*)

Return a list of edges.

Edges are returned as tuples with optional data and keys in the order (node, neighbor, key, data).

Parameters **nbunch** : iterable container, optional (default= all nodes)

A container of nodes. The container will be iterated through once.

data : bool, optional (default=False)

Return two tuples (u,v) (False) or three-tuples (u,v,data) (True).

keys : bool, optional (default=False)

Return two tuples (u,v) (False) or three-tuples (u,v,key) (True).

Returns **edge_list: list of edge tuples** :

Edges that are adjacent to any node in nbunch, or a list of all edges if nbunch is not specified.

See Also:

`edges_iter` return an iterator over the edges

Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored.

Examples

```
>>> G = nx.MultiGraph()  # or MultiDiGraph
>>> G.add_path([0,1,2,3])
>>> G.edges()
[(0, 1), (1, 2), (2, 3)]
>>> G.edges(data=True) # default edge data is {} (empty dictionary)
[(0, 1, {}), (1, 2, {}), (2, 3, {})]
>>> G.edges(keys=True) # default keys are integers
[(0, 1, 0), (1, 2, 0), (2, 3, 0)]
>>> G.edges(data=True,keys=True) # default keys are integers
[(0, 1, 0, {}), (1, 2, 0, {}), (2, 3, 0, {})]
>>> G.edges([0,3])
[(0, 1), (3, 2)]
>>> G.edges(0)
[(0, 1)]
```

networkx.MultiGraph.edges_iter

MultiGraph.**edges_iter**(*nbunch=None*, *data=False*, *keys=False*)

Return an iterator over the edges.

Edges are returned as tuples with optional data and keys in the order (node, neighbor, key, data).

Parameters **nbunch** : iterable container, optional (default= all nodes)

A container of nodes. The container will be iterated through once.

data : bool, optional (default=False)

If True, return edge attribute dict with each edge.

keys : bool, optional (default=False)

If True, return edge keys with each edge.

Returns **edge_iter** : iterator

An iterator of (u,v), (u,v,d) or (u,v,key,d) tuples of edges.

See Also:

edges return a list of edges

Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored.

Examples

```
>>> G = nx.MultiGraph()    # or MultiDiGraph
>>> G.add_path([0,1,2,3])
>>> [e for e in G.edges_iter()]
[(0, 1), (1, 2), (2, 3)]
>>> list(G.edges_iter(data=True)) # default data is {} (empty dict)
[(0, 1, {}), (1, 2, {}), (2, 3, {})]
>>> list(G.edges(keys=True)) # default keys are integers
[(0, 1, 0), (1, 2, 0), (2, 3, 0)]
>>> list(G.edges(data=True,keys=True)) # default keys are integers
[(0, 1, 0, {}), (1, 2, 0, {}), (2, 3, 0, {})]
>>> list(G.edges_iter([0,3]))
[(0, 1), (3, 2)]
>>> list(G.edges_iter(0))
[(0, 1)]
```

networkx.MultiGraph.get_edge_data

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

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

Parameters **u,v** : nodes

default: any Python object (default=None) :

Value to return if the edge (u,v) is not found.

key : hashable identifier, optional (default=None)

Return data only for the edge with specified key.

Returns **edge_dict** : dictionary

The edge attribute dictionary.

Notes

It is faster to use G[u][v][key].

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

Warning: Assigning G[u][v][key] corrupts the graph data structure. But it is safe to assign attributes to that dictionary,

```
>>> G[0][1]['a']['weight'] = 10
>>> G[0][1]['a']['weight']
10
>>> G[1][0]['a']['weight']
10
```

Examples

```
>>> G = nx.MultiGraph() # or MultiDiGraph
>>> G.add_path([0,1,2,3])
>>> G.get_edge_data(0,1)
{0: {}}
>>> e = (0,1)
>>> G.get_edge_data(*e) # tuple form
{0: {}}
>>> G.get_edge_data('a','b',default=0) # edge not in graph, return 0
0
```

networkx.MultiGraph.neighbors

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

Return a list of the nodes connected to the node n.

Parameters **n** : node

A node in the graph

Returns **nlist** : list

A list of nodes that are adjacent to n.

Raises **NetworkXError** :

If the node n is not in the graph.

Notes

It is usually more convenient (and faster) to access the adjacency dictionary as G[n]:

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge('a','b',weight=7)
>>> G['a']
{'b': {'weight': 7}}
```

Examples

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

networkx.MultiGraph.neighbors_iter

MultiGraph.**neighbors_iter**(*n*)

Return an iterator over all neighbors of node n.

Notes

It is faster to use the idiom "in G[0]", e.g.

```
>>> G = nx.path_graph(4)
>>> [n for n in G[0]]
[1]
```

Examples

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

networkx.MultiGraph.__getitem__

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

Return a dict of neighbors of node n. Use the expression '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 similar to G.neighbors(n) but the internal data dictionary is returned instead of a list.

Assigning G[n] will corrupt the internal graph data structure. Use G[n] for reading data only.

Examples

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G[0]
{1: {}}
```

networkx.MultiGraph.adjacency_list

MultiGraph.**adjacency_list**()

Return an adjacency list representation of the graph.

The output adjacency list is in the order of G.nodes(). For directed graphs, only outgoing adjacencies are included.

Returns **adj_list** : lists of lists

The adjacency structure of the graph as a list of lists.

See Also:

adjacency_iter

Examples

```
>>> G = nx.Graph()   # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.adjacency_list() # in order given by G.nodes()
[[1], [0, 2], [1, 3], [2]]
```

networkx.MultiGraph.adjacency_iter

MultiGraph.**adjacency_iter**()

Return an iterator of (node, adjacency dict) tuples for all nodes.

This is the fastest way to look at every edge. For directed graphs, only outgoing adjacencies are included.

Returns **adj_iter** : iterator

An iterator of (node, adjacency dictionary) for all nodes in the graph.

See Also:

adjacency_list

Examples

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

networkx.MultiGraph.nbunch_iter

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

Return an iterator of 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** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

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 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.

Information about graph structure

MultiGraph.has_node(n)	Return True if the graph contains the node n.
MultiGraph.__contains__(n)	Return True if n is a node, False otherwise. Use the expression
MultiGraph.has_edge(u, v[, key])	Return True if the graph has an edge between nodes u and v.
MultiGraph.order()	Return the number of nodes in the graph.
MultiGraph.number_of_nodes()	Return the number of nodes in the graph.
MultiGraph.__len__()	Return the number of nodes.
MultiGraph.degree([nbunch, weighted])	Return the degree of a node or nodes.
MultiGraph.degree_iter([nbunch, weighted])	Return an iterator for (node, degree).
MultiGraph.size([weighted])	Return the number of edges.
MultiGraph.number_of_edges([u, v])	Return the number of edges between two nodes.
MultiGraph.nodes_with_selfloops()	Return a list of nodes with self loops.
MultiGraph.selfloop_edges([data, keys])	Return a list of selfloop edges.
MultiGraph.number_of_selfloops()	Return the number of selfloop edges.

networkx.MultiGraph.has_node

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

Return True if the graph contains the node n.

Parameters **n** : node

Examples

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> G.has_node(0)
True
```

It is more readable and simpler to use

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

networkx.MultiGraph.__contains__

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

Return True if n is a node, False otherwise. Use the expression 'n in G'.

Examples

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> 1 in G
True
```

networkx.MultiGraph.has_edge

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

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

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
>>> G.add_path([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')
>>> G.has_edge(0,1,key='a')  # specify key
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 KeyError if 0 not in G
True
```

networkx.MultiGraph.order

MultiGraph.**order**()

Return the number of nodes in the graph.

Returns **nnodes** : int

The number of nodes in the graph.

See Also:

number_of_nodes, __len__

networkx.MultiGraph.number_of_nodes

MultiGraph.**number_of_nodes**()

Return the number of nodes in the graph.

Returns **nnodes** : int

The number of nodes in the graph.

See Also:

order, __len__

Examples

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

networkx.MultiGraph.__len__

MultiGraph.**__len__**()

Return the number of nodes. Use the expression 'len(G)'.

Returns **nnodes** : int

The number of nodes in the graph.

Examples

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> len(G)
4
```

networkx.MultiGraph.degree

MultiGraph.**degree**(*nbunch=None*, *weighted=False*)

Return the degree of a node or nodes.

The node degree is the number of edges adjacent to that node.

Parameters **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

weighted : bool, optional (default=False)

If True return the sum of edge weights adjacent to the node.

Returns **nd** : dictionary, or number

A dictionary with nodes as keys and degree as values or a number if a single node is specified.

Examples

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

networkx.MultiGraph.degree_iter

MultiGraph.**degree_iter**(*nbunch=None*, *weighted=False*)

Return an iterator for (node, degree).

The node degree is the number of edges adjacent to the node.

Parameters **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

weighted : bool, optional (default=False)

If True return the sum of edge weights adjacent to the node.

Returns **nd_iter** : an iterator

The iterator returns two-tuples of (node, degree).

See Also:

degree

Examples

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

networkx.MultiGraph.size

MultiGraph.**size**(*weighted=False*)

Return the number of edges.

Parameters **weighted** : boolean, optional (default=False)

If True return the sum of the edge weights.

Returns **nedges** : int

The number of edges in the graph.

See Also:

number_of_edges

Examples

```
>>> G = nx.Graph()   # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> 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(weighted=True)
6.0
```

networkx.MultiGraph.number_of_edges

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

Return 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.

See Also:

size

Examples

```
>>> G = nx.Graph()   # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.number_of_edges()
3
>>> G.number_of_edges(0,1)
1
>>> e = (0,1)
>>> G.number_of_edges(*e)
1
```

networkx.MultiGraph.nodes_with_selfloops

MultiGraph.**nodes_with_selfloops**()

Return a list of nodes with self loops.

A node with a self loop has an edge with both ends adjacent to that node.

Returns **nodelist** : list

A list of 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)
>>> G.nodes_with_selfloops()
[1]
```

networkx.MultiGraph.selfloop_edges

`MultiGraph.`**`selfloop_edges`**(*data=False*, *keys=False*)

Return a list of selfloop edges.

A selfloop edge has the same node at both ends.

Parameters **data** : bool, optional (default=False)

Return selfloop edges as two tuples (u,v) (data=False) or three-tuples (u,v,data) (data=True)

keys : bool, optional (default=False)

If True, return edge keys with each edge.

Returns **edgelist** : list of edge tuples

A list of all selfloop edges.

See Also:

`selfloop_nodes`, `number_of_selfloops`

Examples

```
>>> G = nx.MultiGraph()    # or MultiDiGraph
>>> G.add_edge(1,1)
>>> G.add_edge(1,2)
>>> G.selfloop_edges()
[(1, 1)]
>>> G.selfloop_edges(data=True)
[(1, 1, {})]
>>> G.selfloop_edges(keys=True)
[(1, 1, 0)]
>>> G.selfloop_edges(keys=True, data=True)
[(1, 1, 0, {})]
```

networkx.MultiGraph.number_of_selfloops

MultiGraph.**number_of_selfloops**()

Return the number of selfloop edges.

A selfloop edge has the same node at both ends.

Returns **nloops** : int

The number of selfloops.

See Also:

selfloop_nodes, selfloop_edges

Examples

```
>>> G=nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge(1,1)
>>> G.add_edge(1,2)
>>> G.number_of_selfloops()
1
```

Making copies and subgraphs

MultiGraph.copy()	Return a copy of the graph.
MultiGraph.to_undirected()	Return an undirected copy of the graph.
MultiGraph.to_directed()	Return a directed representation of the graph.
MultiGraph.subgraph(nbunch)	Return the subgraph induced on nodes in nbunch.

networkx.MultiGraph.copy

MultiGraph.**copy**()

Return a copy of the graph.

Returns **G** : Graph

A copy of the graph.

See Also:

to_directed return a directed copy of the graph.

Notes

This makes a complete copy of the graph including all of the node or edge attributes.

Examples

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> H = G.copy()
```

networkx.MultiGraph.to_undirected

MultiGraph.**to_undirected**()

Return 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=DiGraph(D) which returns a shallow copy of the data.

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

Examples

```
>>> G = nx.Graph()   # or MultiGraph, etc
>>> G.add_path([0,1])
>>> H = G.to_directed()
>>> H.edges()
[(0, 1), (1, 0)]
>>> G2 = H.to_undirected()
>>> G2.edges()
[(0, 1)]
```

networkx.MultiGraph.to_directed

MultiGraph.**to_directed**()

Return a directed representation of the graph.

Returns **G** : MultiDiGraph

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, http://docs.python.org/library/copy.html.

Examples

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

If already directed, return a (deep) copy

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

networkx.MultiGraph.subgraph

MultiGraph.**subgraph**(*nbunch*)

Return the subgraph induced on nodes in nbunch.

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

Parameters **nbunch** : list, iterable

A container of nodes which will be iterated through once.

Returns **G** : Graph

A subgraph of the graph with the same edge attributes.

Notes

The graph, edge or node attributes just point to the original graph. So changes to the node or edge structure will not be reflected in the original graph while changes to the attributes will.

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

If edge attributes are containers, a deep copy can be obtained using: G.subgraph(nbunch).copy()

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)])

Examples

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> H = G.subgraph([0,1,2])
>>> H.edges()
[(0, 1), (1, 2)]
```

3.2.4 MultiDiGraph - Directed graphs with self loops and parallel edges

Overview

networkx.**MultiDiGraph**(*data=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.

Edges are represented as links between nodes with optional key/value attributes.

Parameters **data** : input graph

Data to initialize graph. If 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 NumPy matrix or 2d ndarray, a SciPy sparse matrix, or a PyGraphviz graph.

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.Graph()
>>> H.add_path([0,1,2,3,4,5,6,7,8,9])
>>> 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. 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.

```
>>> G.add_edges_from([(4,5,dict(route=282)), (4,5,dict(route=37))])
>>> G[4]
{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.node

```
>>> G.add_node(1, time='5pm')
>>> G.add_nodes_from([3], time='2pm')
>>> G.node[1]
{'time': '5pm'}
>>> G.node[1]['room'] = 714
>>> G.nodes(data=True)
[(1, {'room': 714, 'time': '5pm'}), (3, {'time': '2pm'})]
```

Warning: adding a node to G.node does not add it to the graph.

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

```
>>> 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][0]['weight'] = 4.7
>>> G.edge[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 keyed by neighbor to edge attributes
...            # Note: you should not change this dict manually!
{2: {0: {'weight': 4}, 1: {'color': 'blue'}}}
```

The fastest way to traverse all edges of a graph is via adjacency_iter(), but the edges() method is often more convenient.

```
>>> for n,nbrsdict in G.adjacency_iter():
...     for nbr,keydict in nbrsdict.items():
...         for key,eattr in keydict.items():
...             if 'weight' in eattr:
...                 (n,nbr,eattr['weight'])
(1, 2, 4)
(2, 3, 8)
>>> [ (u,v,edata['weight']) for u,v,edata in G.edges(data=True) if 'weight' in edata ]
[(1, 2, 4), (2, 3, 8)]
```

Reporting:

Simple graph information is obtained using methods. Iterator versions of many reporting methods exist for efficiency. Methods exist for reporting nodes(), edges(), neighbors() and degree() as well as the number of nodes and edges.

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

Adding and Removing Nodes and Edges

MultiDiGraph.__init__(**attr[, data])	Initialize a graph with edges, name, graph attributes.
MultiDiGraph.add_node(n, **attr[, attr_dict])	Add a single node n and update node attributes.
MultiDiGraph.add_nodes_from(nodes, **attr)	Add multiple nodes.
MultiDiGraph.remove_node(n)	Remove node n.
MultiDiGraph.remove_nodes_from(nbunch)	Remove multiple nodes.
MultiDiGraph.add_edge(u, v, **attr[, key, ...])	Add an edge between u and v.
MultiDiGraph.add_edges_from(ebunch, **attr)	Add all the edges in ebunch.
MultiDiGraph.add_weighted_edges_from(ebunch, ...)	Add all the edges in ebunch as weighted edges with specified weights.
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.add_star(nodes, **attr)	Add a star.
MultiDiGraph.add_path(nodes, **attr)	Add a path.
MultiDiGraph.add_cycle(nodes, **attr)	Add a cycle.
MultiDiGraph.clear()	Remove all nodes and edges from the graph.

networkx.MultiDiGraph.__init__

MultiDiGraph.**__init__**(*data=None*, ***attr*)

Initialize a graph with edges, name, graph attributes.

Parameters **data** : input graph

Data to initialize graph. If 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 NumPy matrix or 2d ndarray, a SciPy sparse matrix, or a PyGraphviz graph.

name : string, optional (default='')

An optional name for the 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'}
```

networkx.MultiDiGraph.add_node

MultiDiGraph.**add_node**(*n*, *attr_dict=None*, ***attr*)

Add a single node n and update node attributes.

Parameters **n** : node

A node can be any hashable Python object except None.

attr_dict : dictionary, optional (default= no attributes)

Dictionary of node attributes. Key/value pairs will update existing data associated with the node.

attr : keyword arguments, optional

Set or change 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))
```

networkx.MultiDiGraph.add_nodes_from

MultiDiGraph.**add_nodes_from**(*nodes*, ***attr*)

Add multiple nodes.

Parameters **nodes** : 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 generally.

See Also:

add_node

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.node[1]['size']
11
>>> H = nx.Graph()
>>> H.add_nodes_from(G.nodes(data=True))
>>> H.node[1]['size']
11
```

networkx.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.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> G.edges()
[(0, 1), (1, 2)]
>>> G.remove_node(1)
>>> G.edges()
[]
```

networkx.MultiDiGraph.remove_nodes_from

MultiDiGraph.**remove_nodes_from**(*nbunch*)

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

Examples

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

networkx.MultiDiGraph.add_edge

MultiDiGraph.**add_edge**(*u*, *v*, *key=None*, *attr_dict=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 providing a dictionary with key/value pairs. See examples below.

Parameters **u,v** : 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_dict : dictionary, optional (default= no attributes)

Dictionary of edge attributes. Key/value pairs will update existing data associated with the edge.

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

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.

Examples

The following all add the edge e=(1,2) to graph G:

```
>>> G = nx.MultiDiGraph()
>>> 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, 2, key=0, weight=4)   # update data for key=0
>>> G.add_edge(1, 3, weight=7, capacity=15, length=342.7)
```

networkx.MultiDiGraph.add_edges_from

MultiDiGraph.**add_edges_from**(*ebunch*, *attr_dict=None*, ***attr*)

Add all the edges in ebunch.

Parameters **ebunch** : 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 attribute dict d, or
- 4-tuples (u,v,k,d) for an edge identified by key k

attr_dict : dictionary, optional (default= no attributes)

Dictionary of edge attributes. Key/value pairs will update existing data associated with each edge.

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.

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')
```

networkx.MultiDiGraph.add_weighted_edges_from

MultiDiGraph.**add_weighted_edges_from**(*ebunch*, ***attr*)

Add all the edges in ebunch as weighted edges with specified weights.

Parameters **ebunch** : 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.

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 has no effect but any edge data will be updated when each duplicate edge is added.

Examples

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

networkx.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 (abritrary) edge between u and v.

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()
>>> G.add_path([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)])
>>> G.remove_edge(1,2) # remove a single (arbitrary) edge
```

For edges with keys

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

networkx.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) All edges between u and v are removed.
- 3-tuples (u,v,key) The edge identified by key is removed.

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.MultiGraph() # or MultiDiGraph
>>> G.add_path([0,1,2,3])
>>> ebunch=[(1,2),(2,3)]
>>> G.remove_edges_from(ebunch)
```

Removing multiple copies of edges

```
>>> G = nx.MultiGraph()
>>> G.add_edges_from([(1,2),(1,2),(1,2)])
>>> G.remove_edges_from([(1,2),(1,2)])
>>> G.edges()
[(1, 2)]
>>> G.remove_edges_from([(1,2),(1,2)]) # silently ignore extra copy
>>> G.edges() # now empty graph
[]
```

networkx.MultiDiGraph.add_star

MultiDiGraph.**add_star**(*nodes*, ***attr*)

Add a star.

The first node in nodes is the middle of the star. It is connected to all other nodes.

Parameters **nodes** : 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()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_star([0,1,2,3])
>>> G.add_star([10,11,12],weight=2)
```

networkx.MultiDiGraph.add_path

MultiDiGraph.**add_path**(*nodes*, ***attr*)

Add a path.

Parameters **nodes** : 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()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.add_path([10,11,12],weight=7)
```

networkx.MultiDiGraph.add_cycle

MultiDiGraph.**add_cycle**(*nodes*, ***attr*)

Add a cycle.

Parameters **nodes: 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
>>> G.add_cycle([0,1,2,3])
>>> G.add_cycle([10,11,12],weight=7)
```

networkx.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.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.clear()
>>> G.nodes()
[]
>>> G.edges()
[]
```

Iterating over nodes and edges

MultiDiGraph.nodes([data])	Return a list of the nodes in the graph.
MultiDiGraph.nodes_iter([data])	Return an iterator over the nodes.
MultiDiGraph.__iter__()	Iterate over the nodes.
MultiDiGraph.edges([nbunch, data, keys])	Return a list of edges.
MultiDiGraph.edges_iter([nbunch, data, keys])	Return an iterator over the edges.
MultiDiGraph.out_edges([nbunch, keys, data])	Return a list of the outgoing edges.
MultiDiGraph.out_edges_iter([nbunch, data, keys])	Return an iterator over the edges.
MultiDiGraph.in_edges([nbunch, keys, data])	Return a list of the incoming edges.
MultiDiGraph.in_edges_iter([nbunch, data, keys])	Return an iterator over the incoming edges.
MultiDiGraph.get_edge_data(u, v[, key, default])	Return the attribute dictionary associated with edge (u,v).
MultiDiGraph.neighbors(n)	Return a list of successor nodes of n.
MultiDiGraph.neighbors_iter(n)	Return an iterator over successor nodes of n.
MultiDiGraph.__getitem__(n)	Return a dict of neighbors of node n.
MultiDiGraph.successors(n)	Return a list of successor nodes of n.
MultiDiGraph.successors_iter(n)	Return an iterator over successor nodes of n.
MultiDiGraph.predecessors(n)	Return a list of predecessor nodes of n.
MultiDiGraph.predecessors_iter(n)	Return an iterator over predecessor nodes of n.
MultiDiGraph.adjacency_list()	Return an adjacency list representation of the graph.
MultiDiGraph.adjacency_iter()	Return an iterator of (node, adjacency dict) tuples for all nodes.
MultiDiGraph.nbunch_iter([nbunch])	Return an iterator of nodes contained in nbunch that are also in the graph.

networkx.MultiDiGraph.nodes

MultiDiGraph.**nodes**(*data=False*)
Return a list of the nodes in the graph.

Parameters **data** : boolean, optional (default=False)

If False return a list of nodes. If True return a two-tuple of node and node data dictionary

Returns **nlist** : list

A list of nodes. If data=True a list of two-tuples containing (node, node data dictionary).

Examples

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> G.nodes()
[0, 1, 2]
>>> G.add_node(1, time='5pm')
>>> G.nodes(data=True)
[(0, {}), (1, {'time': '5pm'}), (2, {})]
```

networkx.MultiDiGraph.nodes_iter

MultiDiGraph.**nodes_iter**(*data=False*)

Return an iterator over the nodes.

Parameters **data** : boolean, optional (default=False)

If False the iterator returns nodes. If True return a two-tuple of node and node data dictionary

Returns **niter** : iterator

An iterator over nodes. If data=True the iterator gives two-tuples containing (node, node data, dictionary)

Notes

If the node data is not required it is simpler and equivalent to use the expression 'for n in G'.

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

Examples

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

>>> [d for n,d in G.nodes_iter(data=True)]
[{}, {}, {}]
```

networkx.MultiDiGraph.__iter__

MultiDiGraph.**__iter__**()

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

Returns **niter** : iterator

An iterator over all nodes in the graph.

Examples

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

networkx.MultiDiGraph.edges

MultiDiGraph.**edges**(*nbunch=None*, *data=False*, *keys=False*)

Return a list of edges.

Edges are returned as tuples with optional data and keys in the order (node, neighbor, key, data).

Parameters **nbunch** : iterable container, optional (default= all nodes)

A container of nodes. The container will be iterated through once.

data : bool, optional (default=False)

Return two tuples (u,v) (False) or three-tuples (u,v,data) (True).

keys : bool, optional (default=False)

Return two tuples (u,v) (False) or three-tuples (u,v,key) (True).

Returns **edge_list: list of edge tuples** :

Edges that are adjacent to any node in nbunch, or a list of all edges if nbunch is not specified.

See Also:

`edges_iter` return an iterator over the edges

Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored.

Examples

```
>>> G = nx.MultiGraph()  # or MultiDiGraph
>>> G.add_path([0,1,2,3])
>>> G.edges()
[(0, 1), (1, 2), (2, 3)]
>>> G.edges(data=True) # default edge data is {} (empty dictionary)
[(0, 1, {}), (1, 2, {}), (2, 3, {})]
>>> G.edges(keys=True) # default keys are integers
[(0, 1, 0), (1, 2, 0), (2, 3, 0)]
>>> G.edges(data=True,keys=True) # default keys are integers
[(0, 1, 0, {}), (1, 2, 0, {}), (2, 3, 0, {})]
>>> G.edges([0,3])
[(0, 1), (3, 2)]
>>> G.edges(0)
[(0, 1)]
```

networkx.MultiDiGraph.edges_iter

MultiDiGraph.**edges_iter**(*nbunch=None*, *data=False*, *keys=False*)

Return an iterator over the edges.

Edges are returned as tuples with optional data and keys in the order (node, neighbor, key, data).

Parameters **nbunch** : iterable container, optional (default= all nodes)

A container of nodes. The container will be iterated through once.

data : bool, optional (default=False)

If True, return edge attribute dict with each edge.

keys : bool, optional (default=False)

If True, return edge keys with each edge.

Returns **edge_iter** : iterator

An iterator of (u,v), (u,v,d) or (u,v,key,d) tuples of edges.

See Also:

`edges` return a list of edges

Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored. For directed graphs edges() is the same as out_edges().

Examples

```
>>> G = nx.MultiDiGraph()
>>> G.add_path([0,1,2,3])
>>> [e for e in G.edges_iter()]
[(0, 1), (1, 2), (2, 3)]
>>> list(G.edges_iter(data=True)) # default data is {} (empty dict)
[(0, 1, {}), (1, 2, {}), (2, 3, {})]
>>> list(G.edges_iter([0,2]))
[(0, 1), (2, 3)]
>>> list(G.edges_iter(0))
[(0, 1)]
```

networkx.MultiDiGraph.out_edges

MultiDiGraph.**out_edges**(*nbunch=None*, *keys=False*, *data=False*)

Return a list of the outgoing edges.

Edges are returned as tuples with optional data and keys in the order (node, neighbor, key, data).

Parameters **nbunch** : iterable container, optional (default= all nodes)

A container of nodes. The container will be iterated through once.

data : bool, optional (default=False)

If True, return edge attribute dict with each edge.

keys : bool, optional (default=False)

If True, return edge keys with each edge.

Returns **out_edges** : list

An listr of (u,v), (u,v,d) or (u,v,key,d) tuples of edges.

See Also:

`in_edges` return a list of incoming edges

Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored. For directed graphs edges() is the same as out_edges().

networkx.MultiDiGraph.out_edges_iter

MultiDiGraph.**out_edges_iter**(*nbunch=None*, *data=False*, *keys=False*)

Return an iterator over the edges.

Edges are returned as tuples with optional data and keys in the order (node, neighbor, key, data).

Parameters **nbunch** : iterable container, optional (default= all nodes)

A container of nodes. The container will be iterated through once.

data : bool, optional (default=False)

If True, return edge attribute dict with each edge.

keys : bool, optional (default=False)

If True, return edge keys with each edge.

Returns **edge_iter** : iterator

An iterator of (u,v), (u,v,d) or (u,v,key,d) tuples of edges.

See Also:

`edges` return a list of edges

Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored. For directed graphs edges() is the same as out_edges().

Examples

```
>>> G = nx.MultiDiGraph()
>>> G.add_path([0,1,2,3])
>>> [e for e in G.edges_iter()]
[(0, 1), (1, 2), (2, 3)]
>>> list(G.edges_iter(data=True)) # default data is {} (empty dict)
[(0, 1, {}), (1, 2, {}), (2, 3, {})]
```

```
>>> list(G.edges_iter([0,2]))
[(0, 1), (2, 3)]
>>> list(G.edges_iter(0))
[(0, 1)]
```

networkx.MultiDiGraph.in_edges

MultiDiGraph.**in_edges**(*nbunch=None*, *keys=False*, *data=False*)

Return a list of the incoming edges.

Parameters **nbunch** : iterable container, optional (default= all nodes)

A container of nodes. The container will be iterated through once.

data : bool, optional (default=False)

If True, return edge attribute dict with each edge.

keys : bool, optional (default=False)

If True, return edge keys with each edge.

Returns **in_edges** : list

A list of (u,v), (u,v,d) or (u,v,key,d) tuples of edges.

See Also:

out_edges return a list of outgoing edges

networkx.MultiDiGraph.in_edges_iter

MultiDiGraph.**in_edges_iter**(*nbunch=None*, *data=False*, *keys=False*)

Return an iterator over the incoming edges.

Parameters **nbunch** : iterable container, optional (default= all nodes)

A container of nodes. The container will be iterated through once.

data : bool, optional (default=False)

If True, return edge attribute dict with each edge.

keys : bool, optional (default=False)

If True, return edge keys with each edge.

Returns **in_edge_iter** : iterator

An iterator of (u,v), (u,v,d) or (u,v,key,d) tuples of edges.

See Also:

edges_iter return an iterator of edges

networkx.MultiDiGraph.get_edge_data

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

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

Parameters **u,v** : nodes

default: any Python object (default=None) :

Value to return if the edge (u,v) is not found.

key : hashable identifier, optional (default=None)

Return data only for the edge with specified key.

Returns **edge_dict** : dictionary

The edge attribute dictionary.

Notes

It is faster to use G[u][v][key].

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

Warning: Assigning G[u][v][key] corrupts the graph data structure. But it is safe to assign attributes to that dictionary,

```
>>> G[0][1]['a']['weight'] = 10
>>> G[0][1]['a']['weight']
10
>>> G[1][0]['a']['weight']
10
```

Examples

```
>>> G = nx.MultiGraph() # or MultiDiGraph
>>> G.add_path([0,1,2,3])
>>> G.get_edge_data(0,1)
{0: {}}
>>> e = (0,1)
>>> G.get_edge_data(*e) # tuple form
{0: {}}
>>> G.get_edge_data('a','b',default=0) # edge not in graph, return 0
0
```

networkx.MultiDiGraph.neighbors

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

Return a list of successor nodes of n.

neighbors() and successors() are the same function.

networkx.MultiDiGraph.neighbors_iter

MultiDiGraph.**neighbors_iter**(*n*)

Return an iterator over successor nodes of n.

neighbors_iter() and successors_iter() are the same.

networkx.MultiDiGraph.__getitem__

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

Return a dict of neighbors of node n. Use the expression '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 similar to G.neighbors(n) but the internal data dictionary is returned instead of a list.

Assigning G[n] will corrupt the internal graph data structure. Use G[n] for reading data only.

Examples

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G[0]
{1: {}}
```

networkx.MultiDiGraph.successors

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

Return a list of successor nodes of n.

neighbors() and successors() are the same function.

networkx.MultiDiGraph.successors_iter

MultiDiGraph.**successors_iter**(*n*)

Return an iterator over successor nodes of n.

neighbors_iter() and successors_iter() are the same.

networkx.MultiDiGraph.predecessors

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

Return a list of predecessor nodes of n.

networkx.MultiDiGraph.predecessors_iter

MultiDiGraph.**predecessors_iter**(*n*)

Return an iterator over predecessor nodes of n.

networkx.MultiDiGraph.adjacency_list

MultiDiGraph.**adjacency_list**()

Return an adjacency list representation of the graph.

The output adjacency list is in the order of G.nodes(). For directed graphs, only outgoing adjacencies are included.

Returns **adj_list** : lists of lists

The adjacency structure of the graph as a list of lists.

See Also:

adjacency_iter

Examples

```
>>> G = nx.Graph()   # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.adjacency_list() # in order given by G.nodes()
[[1], [0, 2], [1, 3], [2]]
```

networkx.MultiDiGraph.adjacency_iter

MultiDiGraph.**adjacency_iter**()

Return an iterator of (node, adjacency dict) tuples for all nodes.

This is the fastest way to look at every edge. For directed graphs, only outgoing adjacencies are included.

Returns **adj_iter** : iterator

An iterator of (node, adjacency dictionary) for all nodes in the graph.

See Also:

adjacency_list

Examples

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

networkx.MultiDiGraph.nbunch_iter

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

Return an iterator of 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** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

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 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.

Information about graph structure

`MultiDiGraph.has_node`(n)	Return True if the graph contains the node n.
`MultiDiGraph.__contains__`(n)	Return True if n is a node, False otherwise. Use the expression
`MultiDiGraph.has_edge`(u, v[, key])	Return True if the graph has an edge between nodes u and v.
`MultiDiGraph.order`()	Return the number of nodes in the graph.
`MultiDiGraph.number_of_nodes`()	Return the number of nodes in the graph.
`MultiDiGraph.__len__`()	Return the number of nodes.
`MultiDiGraph.degree`([nbunch, weighted])	Return the degree of a node or nodes.
`MultiDiGraph.degree_iter`([nbunch, weighted])	Return an iterator for (node, degree).
`MultiDiGraph.in_degree`([nbunch, weighted])	Return the in-degree of a node or nodes.
`MultiDiGraph.in_degree_iter`([nbunch, weighted])	Return an iterator for (node, in-degree).
`MultiDiGraph.out_degree`([nbunch, weighted])	Return the out-degree of a node or nodes.
`MultiDiGraph.out_degree_iter`([nbunch, weighted])	Return an iterator for (node, out-degree).
`MultiDiGraph.size`([weighted])	Return the number of edges.
`MultiDiGraph.number_of_edges`([u, v])	Return the number of edges between two nodes.
`MultiDiGraph.nodes_with_selfloops`()	Return a list of nodes with self loops.
`MultiDiGraph.selfloop_edges`([data, keys])	Return a list of selfloop edges.
`MultiDiGraph.number_of_selfloops`()	Return the number of selfloop edges.

networkx.MultiDiGraph.has_node

`MultiDiGraph.`**`has_node`**`(`*`n`*`)`

Return True if the graph contains the node n.

Parameters **n** : node

Examples

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> G.has_node(0)
True
```

It is more readable and simpler to use

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

networkx.MultiDiGraph.__contains__

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

Return True if n is a node, False otherwise. Use the expression 'n in G'.

Examples

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> 1 in G
True
```

networkx.MultiDiGraph.has_edge

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

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

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
>>> G.add_path([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')
>>> G.has_edge(0,1,key='a')  # specify key
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 KeyError if 0 not in G
True
```

networkx.MultiDiGraph.order

MultiDiGraph.**order**()

Return the number of nodes in the graph.

Returns **nnodes** : int

The number of nodes in the graph.

See Also:

number_of_nodes, __len__

networkx.MultiDiGraph.number_of_nodes

MultiDiGraph.**number_of_nodes**()

Return the number of nodes in the graph.

Returns **nnodes** : int

The number of nodes in the graph.

See Also:

order, __len__

Examples

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

networkx.MultiDiGraph.__len__

MultiDiGraph.**__len__**()

Return the number of nodes. Use the expression 'len(G)'.

Returns **nnodes** : int

The number of nodes in the graph.

Examples

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> len(G)
4
```

networkx.MultiDiGraph.degree

MultiDiGraph.**degree**(*nbunch=None*, *weighted=False*)

Return the degree of a node or nodes.

The node degree is the number of edges adjacent to that node.

Parameters **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

weighted : bool, optional (default=False)

If True return the sum of edge weights adjacent to the node.

Returns **nd** : dictionary, or number

A dictionary with nodes as keys and degree as values or a number if a single node is specified.

Examples

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

networkx.MultiDiGraph.degree_iter

MultiDiGraph.**degree_iter**(*nbunch=None*, *weighted=False*)

Return an iterator for (node, degree).

The node degree is the number of edges adjacent to the node.

Parameters **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

weighted : bool, optional (default=False)

If True return the sum of edge weights adjacent to the node.

Returns **nd_iter** : an iterator

The iterator returns two-tuples of (node, degree).

See Also:

degree

Examples

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

networkx.MultiDiGraph.in_degree

MultiDiGraph.**in_degree**(*nbunch=None*, *weighted=False*)

Return the in-degree of a node or nodes.

The node in-degree is the number of edges pointing in to the node.

Parameters **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

weighted : bool, optional (default=False)

If True return the sum of edge weights adjacent to the node.

Returns **nd** : dictionary, or number

A dictionary with nodes as keys and in-degree as values or a number if a single node is specified.

See Also:

degree, out_degree, in_degree_iter

Examples

```
>>> G = nx.DiGraph()    # or MultiDiGraph
>>> G.add_path([0,1,2,3])
>>> G.in_degree(0)
0
>>> G.in_degree([0,1])
{0: 0, 1: 1}
>>> list(G.in_degree([0,1]).values())
[0, 1]
```

networkx.MultiDiGraph.in_degree_iter

MultiDiGraph.**in_degree_iter**(*nbunch=None*, *weighted=False*)

Return an iterator for (node, in-degree).

The node in-degree is the number of edges pointing in to the node.

Parameters **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

weighted : bool, optional (default=False)

If True return the sum of edge weights adjacent to the node.

Returns **nd_iter** : an iterator

The iterator returns two-tuples of (node, in-degree).

See Also:

degree, in_degree, out_degree, out_degree_iter

Examples

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

networkx.MultiDiGraph.out_degree

MultiDiGraph.**out_degree**(*nbunch=None*, *weighted=False*)

Return the out-degree of a node or nodes.

The node out-degree is the number of edges pointing out of the node.

Parameters **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

weighted : bool, optional (default=False)

If True return the sum of edge weights adjacent to the node.

Returns **nd** : dictionary, or number

A dictionary with nodes as keys and out-degree as values or a number if a single node is specified.

Examples

```
>>> G = nx.DiGraph()    # or MultiDiGraph
>>> G.add_path([0,1,2,3])
>>> G.out_degree(0)
1
>>> G.out_degree([0,1])
{0: 1, 1: 1}
>>> list(G.out_degree([0,1]).values())
[1, 1]
```

networkx.MultiDiGraph.out_degree_iter

MultiDiGraph.**out_degree_iter**(*nbunch=None*, *weighted=False*)

Return an iterator for (node, out-degree).

The node out-degree is the number of edges pointing out of the node.

Parameters **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

weighted : bool, optional (default=False)

If True return the sum of edge weights adjacent to the node.

Returns **nd_iter** : an iterator

The iterator returns two-tuples of (node, out-degree).

See Also:

degree, in_degree, out_degree, in_degree_iter

Examples

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

networkx.MultiDiGraph.size

MultiDiGraph.**size**(*weighted=False*)

Return the number of edges.

Parameters **weighted** : boolean, optional (default=False)

If True return the sum of the edge weights.

Returns **nedges** : int

The number of edges in the graph.

See Also:

number_of_edges

Examples

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> 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(weighted=True)
6.0
```

networkx.MultiDiGraph.number_of_edges

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

Return 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.

See Also:

size

Examples

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.number_of_edges()
3
>>> G.number_of_edges(0,1)
1
>>> e = (0,1)
>>> G.number_of_edges(*e)
1
```

networkx.MultiDiGraph.nodes_with_selfloops

MultiDiGraph.**nodes_with_selfloops**()

Return a list of nodes with self loops.

A node with a self loop has an edge with both ends adjacent to that node.

Returns **nodelist** : list

A list of 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)
>>> G.nodes_with_selfloops()
[1]
```

networkx.MultiDiGraph.selfloop_edges

MultiDiGraph.**selfloop_edges**(*data=False*, *keys=False*)

Return a list of selfloop edges.

A selfloop edge has the same node at both ends.

Parameters **data** : bool, optional (default=False)

Return selfloop edges as two tuples (u,v) (data=False) or three-tuples (u,v,data) (data=True)

keys : bool, optional (default=False)

If True, return edge keys with each edge.

Returns **edgelist** : list of edge tuples

A list of all selfloop edges.

See Also:

selfloop_nodes, number_of_selfloops

Examples

```
>>> G = nx.MultiGraph()    # or MultiDiGraph
>>> G.add_edge(1,1)
>>> G.add_edge(1,2)
>>> G.selfloop_edges()
[(1, 1)]
>>> G.selfloop_edges(data=True)
[(1, 1, {})]
>>> G.selfloop_edges(keys=True)
[(1, 1, 0)]
>>> G.selfloop_edges(keys=True, data=True)
[(1, 1, 0, {})]
```

networkx.MultiDiGraph.number_of_selfloops

MultiDiGraph.**number_of_selfloops**()

Return the number of selfloop edges.

A selfloop edge has the same node at both ends.

Returns **nloops** : int

The number of selfloops.

See Also:

selfloop_nodes, selfloop_edges

Examples

```
>>> G=nx.Graph()   # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge(1,1)
>>> G.add_edge(1,2)
>>> G.number_of_selfloops()
1
```

Making copies and subgraphs

MultiDiGraph.copy()	Return a copy of the graph.
MultiDiGraph.to_undirected([reciprocal])	Return an undirected representation of the digraph.
MultiDiGraph.to_directed()	Return a directed copy of the graph.
MultiDiGraph.subgraph(nbunch)	Return the subgraph induced on nodes in nbunch.
MultiDiGraph.reverse([copy])	Return the reverse of the graph.

networkx.MultiDiGraph.copy

MultiDiGraph.**copy**()

Return a copy of the graph.

Returns **G** : Graph

A copy of the graph.

See Also:

to_directed return a directed copy of the graph.

Notes

This makes a complete copy of the graph including all of the node or edge attributes.

Examples

```
>>> G = nx.Graph()   # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> H = G.copy()
```

networkx.MultiDiGraph.to_undirected

MultiDiGraph.**to_undirected**(*reciprocal=False*)

Return an undirected representation of the digraph.

Parameters **reciprocal** : bool (optional)

If True only keep edges that appear in both directions in the original digraph.

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.

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, http://docs.python.org/library/copy.html.

networkx.MultiDiGraph.to_directed

`MultiDiGraph.`**`to_directed`**`()`

Return a directed copy of the graph.

Returns **G** : MultiDiGraph

A deepcopy of the graph.

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, http://docs.python.org/library/copy.html.

Examples

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

If already directed, return a (deep) copy

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

networkx.MultiDiGraph.subgraph

MultiDiGraph.**subgraph**(*nbunch*)

Return the subgraph induced on nodes in nbunch.

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

Parameters **nbunch** : list, iterable

A container of nodes which will be iterated through once.

Returns **G** : Graph

A subgraph of the graph with the same edge attributes.

Notes

The graph, edge or node attributes just point to the original graph. So changes to the node or edge structure will not be reflected in the original graph while changes to the attributes will.

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

If edge attributes are containers, a deep copy can be obtained using: G.subgraph(nbunch).copy()

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)])

Examples

```
>>> G = nx.Graph()   # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> H = G.subgraph([0,1,2])
>>> H.edges()
[(0, 1), (1, 2)]
```

networkx.MultiDiGraph.reverse

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

Return 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, reverse the reverse graph is created using the original graph (this changes the original graph).

CHAPTER

FOUR

ALGORITHMS

4.1 Bipartite

This module provides functions and operations for bipartite graphs. Bipartite graphs G(X,Y,E) have two node sets X,Y and edges in E that only connect nodes from opposite sets.

NetworkX does not have a custom bipartite graph class but the Graph() or DiGraph() classes can be used to represent bipartite graphs.

For example:‘

```
>>> import networkx as nx
>>> top_nodes=[1,1,2,3,3]
>>> bottom_nodes=['a','b','b','b','c']
>>> edges=zip(top_nodes,bottom_nodes) # create 2-tuples of edges
>>> B=nx.Graph(edges)
>>> print(B.edges())
[('a', 1), (1, 'b'), (2, 'b'), ('b', 3), ('c', 3)]
```

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
```

Some of the functions such as bipartite_density and projected_graph take a node set as an argument in addition to the graph B.

```
>>> print(bipartite.density(B,top_nodes))
1.0
>>> G=bipartite.projected_graph(B,bottom_nodes)
>>> G.edges()
[('a', 'b'), ('c', 'b')]
```

You can find the bipartite node sets using

```
>>> X,Y=bipartite.sets(B)
>>> print(list(X))
['a', 'c', 'b']
>>> print(list(Y))
[1, 2, 3]
```

4.1.1 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)	Returns bipartite node sets of graph G.
color(G)	Returns a two-coloring of the graph.
density(B, nodes)	Return density of bipartite graph B.
degrees(B, nodes[, weighted])	Return the degrees of the two node sets in the bipartite graph B.

networkx.algorithms.bipartite.basic.is_bipartite

networkx.algorithms.bipartite.basic.**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
```

networkx.algorithms.bipartite.basic.is_bipartite_node_set

networkx.algorithms.bipartite.basic.**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

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
```

networkx.algorithms.bipartite.basic.sets

networkx.algorithms.bipartite.basic.**sets**(*G*)

Returns bipartite node sets of graph G.

Raises an exception if the graph is not bipartite.

Parameters **G** : NetworkX graph

Returns **(X,Y)** : two-tuple of sets

One set of nodes for each part of the bipartite graph.

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]
```

networkx.algorithms.bipartite.basic.color

networkx.algorithms.bipartite.basic.**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, 'bipartite', c)
>>> print(G.node[0]['bipartite'])
1
>>> print(G.node[1]['bipartite'])
0
```

networkx.algorithms.bipartite.basic.density

networkx.algorithms.bipartite.basic.**density**(*B*, *nodes*)

Return density of bipartite graph B.

Parameters **G** : NetworkX graph

nodes: list or container :

Nodes in one set of the bipartite graph.

Returns **d** : float

The bipartite density

See Also:

color

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
```

networkx.algorithms.bipartite.basic.degrees

networkx.algorithms.bipartite.basic.**degrees**(*B*, *nodes*, *weighted=False*)

Return the degrees of the two node sets in the bipartite graph B.

Parameters **G** : NetworkX graph

nodes: list or container :

Nodes in one set of the bipartite graph.

Returns **(degX,degY)** : tuple of dictionaries

The degrees of the two bipartite sets as dictionaries keyed by node.

See Also:

color, density

Examples

```
>>> from networkx.algorithms import bipartite
>>> G = nx.complete_bipartite_graph(3,2)
>>> Y=set([3,4])
>>> degX,degY=bipartite.degrees(G,Y)
>>> degX
{0: 2, 1: 2, 2: 2}
```

4.1.2 Projections

Create one-mode (unipartite) projections from bipartite graphs.

projected_graph(B, nodes[, multigraph])	Return the graph that is the projection of the bipartite graph B
weighted_projected_graph(B, nodes[, ratio])	Return a weighted unipartite projection of B onto the nodes of
collaboration_weighted_projected_graph(B, nodes)	Weighted unipartite projection of B onto the nodes of
overlap_weighted_projected_graph(B, nodes[, ...])	Return the overlap weighted projection of B onto the nodes of
generic_weighted_projected_graph(B, nodes[, ...])	Return the weighted unipartite projection of B onto the nodes of

networkx.algorithms.bipartite.projection.projected_graph

networkx.algorithms.bipartite.projection.**projected_graph**(*B*, *nodes*, *multigraph=False*)

Return the graph that is the projection of the bipartite graph B onto the specified nodes.

The nodes retain their names and are connected in the resulting graph if 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).

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:

networkx.algorithms.bipartite.basic.is_bipartite, networkx.algorithms.bipartite.basic.is
networkx.algorithms.bipartite.basic.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.

Examples

```
>>> from networkx.algorithms import bipartite
>>> B = nx.path_graph(4)
>>> G = bipartite.projected_graph(B, [1,3])
>>> print(G.nodes())
[1, 3]
>>> print(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(G.edges(keys=True))
[('a', 'b', 1), ('a', 'b', 2)]
```

networkx.algorithms.bipartite.projection.weighted_projected_graph

networkx.algorithms.bipartite.projection.**weighted_projected_graph**(*B*, *nodes*, *ratio=False*)

Return a weighted unipartite projection of B onto the nodes of one bipartite node set.

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=True [R82]. The nodes retain their names 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).

ratio: Bool (default=False) :

If True, edge weight is the ratio between actual shared neighbors and possible shared neighbors. 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:

networkx.algorithms.bipartite.basic.is_bipartite, networkx.algorithms.bipartite.basic.is
networkx.algorithms.bipartite.basic.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. The graph and node properties are (shallow) copied to the projected graph.

References

[R82]

Examples

```
>>> from networkx.algorithms import bipartite
>>> B = nx.path_graph(4)
>>> G = bipartite.weighted_projected_graph(B, [1,3])
>>> print(G.nodes())
[1, 3]
>>> print(G.edges(data=True))
[(1, 3, {'weight': 1})]
>>> G = bipartite.weighted_projected_graph(B, [1,3], ratio=True)
>>> print(G.edges(data=True))
[(1, 3, {'weight': 0.5})]
```

networkx.algorithms.bipartite.projection.collaboration_weighted_projected_graph

networkx.algorithms.bipartite.projection.**collaboration_weighted_projected_graph**(*B*, *nodes*)

Weighted unipartite projection of B onto the nodes of one bipartite node set using the collaboration model.

The collaboration weighted projection is the projection of the bipartite network B onto the specified nodes with weights assigned using Newman's collaboration model [R80]:

..math:

```
w_{v,u} = \sum_k \frac{\delta_{v}^{w} \delta_{w}^{k}}{k_w - 1}
```

where v and u are nodes from the same bipartite node set, and w is a node of the opposite node set. The value k_w is the degree of node w in the bipartite network and δ_v^w is 1 if node v is linked to node w in the original bipartite graph or 0 otherwise.

The nodes retain their names 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:

networkx.algorithms.bipartite.basic.is_bipartite, networkx.algorithms.bipartite.basic.is
networkx.algorithms.bipartite.basic.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.

References

[R80]

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])
>>> print(G.nodes())
[0, 2, 4, 5]
>>> for edge in 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})
```

networkx.algorithms.bipartite.projection.overlap_weighted_projected_graph

networkx.algorithms.bipartite.projection.**overlap_weighted_projected_graph**(*B*, *nodes*, *jaccard=True*)

Return the overlap weighted projection of B onto the nodes of one bipartite node set.

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 [R81]:

..math:

```
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 [R81]:

..math:

```
w_{v,u} = \frac{|N(u) \cap N(v)|}{min(|N(u)|,|N(v)|)}
```

The nodes retain their names 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:

networkx.algorithms.bipartite.basic.is_bipartite, networkx.algorithms.bipartite.basic.is
networkx.algorithms.bipartite.basic.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.

References

[R81]

Examples

```
>>> from networkx.algorithms import bipartite
>>> B = nx.path_graph(5)
>>> G = bipartite.overlap_weighted_projected_graph(B, [0, 2, 4])
>>> print(G.nodes())
[0, 2, 4]
>>> print(G.edges(data=True))
[(0, 2, {'weight': 0.5}), (2, 4, {'weight': 0.5})]
>>> G = bipartite.overlap_weighted_projected_graph(B, [0, 2, 4], jaccard=False)
>>> print(G.edges(data=True))
[(0, 2, {'weight': 1.0}), (2, 4, {'weight': 1.0})]
```

networkx.algorithms.bipartite.projection.generic_weighted_projected_graph

networkx.algorithms.bipartite.projection.**generic_weighted_projected_graph**(*B*, *nodes*, *weight_function=None*)

Return the weighted unipartite projection of B onto the nodes of one bipartite node set 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 names 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 a parameters two sets, the neighborhoods of 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:

networkx.algorithms.bipartite.basic.is_bipartite, networkx.algorithms.bipartite.basic.is
networkx.algorithms.bipartite.basic.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.

Examples

```
>>> from networkx.algorithms import bipartite
>>> def jaccard(unbrs, vnbrs):
...     return float(len(unbrs & vnbrs)) / len(unbrs | vnbrs)
...
>>> def shared(unbrs, vnbrs):
...     return len(unbrs & vnbrs)
...
>>> B = nx.path_graph(5)
>>> G = bipartite.generic_weighted_projected_graph(B, [0, 2, 4], weight_function=jaccard)
>>> print(G.nodes())
[0, 2, 4]
>>> print(G.edges(data=True))
[(0, 2, {'weight': 0.5}), (2, 4, {'weight': 0.5})]
>>> G = bipartite.generic_weighted_projected_graph(B, [0, 2, 4], weight_function=shared)
>>> print(G.nodes())
[0, 2, 4]
>>> print(G.edges(data=True))
[(0, 2, {'weight': 1}), (2, 4, {'weight': 1})]
```

4.1.3 Spectral

Spectral bipartivity measure.

spectral_bipartivity(G[, nodes, weight])	Returns the spectral bipartivity.

networkx.algorithms.bipartite.spectral.spectral_bipartivity

networkx.algorithms.bipartite.spectral.**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:

networkx.algorithms.bipartite.basic.color

Notes

This implementation uses Numpy (dense) matrices which are not efficient for storing large sparse graphs.

References

[R84]

Examples

```
>>> from networkx.algorithms import bipartite
>>> G = nx.path_graph(4)
>>> bipartite.spectral_bipartivity(G)
1.0
```

4.1.4 Clustering

clustering(G[, nodes, mode])	Compute a bipartite clustering coefficient for nodes.
average_clustering(G[, nodes, mode])	Compute the average bipartite clustering coefficient.

networkx.algorithms.bipartite.cluster.clustering

networkx.algorithms.bipartite.cluster.**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 [R79]

$$c_u = \frac{\sum_{v \in N(N(v))} c_{uv}}{|N(N(u))|}$$

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} '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:

networkx.algorithms.bipartite.cluster.average_clustering

References

[R79]

Examples

```
>>> from networkx.algorithms import bipartite
>>> G=nx.path_graph(4) # path is bipartite
>>> c=bipartite.clustering(G)
>>> c[0]
0.5
>>> c=bipartite.clustering(G,mode='min')
>>> c[0]
1.0
```

networkx.algorithms.bipartite.cluster.average_clustering

networkx.algorithms.bipartite.cluster.**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 [R78]

$$C_X = \frac{1}{|X|}\sum_{v \in X} c_v,$$

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:

networkx.algorithms.bipartite.cluster.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.

References

[R78]

Examples

```
>>> from networkx.algorithms import bipartite
>>> G=nx.star_graph(3) # path is 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
```

4.1.5 Redundancy

Node redundancy for bipartite graphs.

node_redundancy(G[, nodes])	Compute bipartite node redundancy coefficient.

networkx.algorithms.bipartite.redundancy.node_redundancy

networkx.algorithms.bipartite.redundancy.**node_redundancy**(*G*, *nodes=None*)

Compute bipartite node redundancy coefficient.

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.

$$rc(v) = \frac{|\{\{u,w\} \subseteq N(v),\ \exists v' \neq v,\ (v',u) \in E \text{ and } (v',w) \in E\}|}{\frac{|N(v)|(|N(v)|-1)}{2}}$$

where $N(v)$ are the 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.

References

[R83]

Examples

```
>>> 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:

```
>>> sum(rc.values())/len(G)
1.0
```

Compute the average redundancy for a set of nodes:

```
>>> nodes = [0, 2]
>>> sum(rc[n] for n in nodes)/len(nodes)
1.0
```

4.1.6 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.

networkx.algorithms.bipartite.centrality.closeness_centrality

networkx.algorithms.bipartite.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, networkx.algorithms.bipartite.basic.sets, networkx.algorithms.bipartite.basic.is_bipartite

Notes

The nodes input parameter must conatin all nodes in one bipartite node set, but the dictionary returned contains all nodes from both node sets.

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 [R76]. 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

[R76]

networkx.algorithms.bipartite.centrality.degree_centrality

networkx.algorithms.bipartite.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, networkx.algorithms.bipartite.basic.sets, networkx.algorithms.bipartite.basic.is_bipartite

Notes

The nodes input parameter must conatin all nodes in one bipartite node set, but the dictionary returned contains all nodes from both bipartite node sets.

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 [R77]. 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

[R77]

networkx.algorithms.bipartite.centrality.betweenness_centrality

networkx.algorithms.bipartite.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 [R75].

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`, `networkx.algorithms.bipartite.basic.sets`, `networkx.algorithms.bipartite.basic.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.

References

[R75]

4.2 Blockmodeling

Functions for creating network blockmodels from node partitions.

`blockmodel`(G, partitions[, multigraph])	Returns a reduced graph constructed using the generalized block modeling technique.

4.2.1 networkx.algorithms.block.blockmodel

networkx.algorithms.block.**blockmodel**(*G*, *partitions*, *multigraph=False*)

Returns a reduced graph constructed using the generalized block modeling technique.

The blockmodel technique collapses nodes into blocks based on a given partitioning of the node set. Each partition of nodes (block) is represented as a single node in the reduced graph.

Edges between nodes in the block graph are added according to the edges in the original graph. If the parameter multigraph is False (the default) a single edge is added with a weight equal to the sum of the edge weights between nodes in the original graph The default is a weight of 1 if weights are not specified. If the parameter multigraph is True then multiple edges are added each with the edge data from the original graph.

Parameters **G** : graph

A networkx Graph or DiGraph

partitions : list of lists, or list of sets

The partition of the nodes. Must be non-overlapping.

multigraph : bool, optional

If True return a MultiGraph with the edge data of the original graph applied to each corresponding edge in the new graph. If False return a Graph with the sum of the edge weights, or a count of the edges if the original graph is unweighted.

Returns **blockmodel** : a Networkx graph object

References

[R85]

Examples

```
>>> G=nx.path_graph(6)
>>> partition=[[0,1],[2,3],[4,5]]
>>> M=nx.blockmodel(G,partition)
```

4.3 Boundary

Routines to find the boundary of a set of nodes.

Edge boundaries are edges that have only one end in the set of nodes.

Node boundaries are nodes outside the set of nodes that have an edge to a node in the set.

edge_boundary(G, nbunch1[, nbunch2])	Return the edge boundary.
node_boundary(G, nbunch1[, nbunch2])	Return the node boundary.

4.3.1 networkx.algorithms.boundary.edge_boundary

networkx.algorithms.boundary.**edge_boundary**(*G*, *nbunch1*, *nbunch2=None*)

Return the edge boundary.

Edge boundaries are edges that have only one end in the given set of nodes.

Parameters **G** : graph

A networkx graph

nbunch1 : list, container

Interior node set

nbunch2 : list, container

Exterior node set. If None then it is set to all of the nodes in G not in nbunch1.

Returns **elist** : list

List of edges

Notes

Nodes in nbunch1 and nbunch2 that are not in G are ignored.

nbunch1 and nbunch2 are usually meant to be disjoint, but in the interest of speed and generality, that is not required here.

4.3.2 networkx.algorithms.boundary.node_boundary

networkx.algorithms.boundary.**node_boundary**(*G*, *nbunch1*, *nbunch2=None*)

Return the node boundary.

The node boundary is all nodes in the edge boundary of a given set of nodes that are in the set.

Parameters **G** : graph

A networkx graph

nbunch1 : list, container

Interior node set

nbunch2 : list, container

Exterior node set. If None then it is set to all of the nodes in G not in nbunch1.

Returns **nlist** : list

List of nodes.

Notes

Nodes in nbunch1 and nbunch2 that are not in G are ignored.

nbunch1 and nbunch2 are usually meant to be disjoint, but in the interest of speed and generality, that is not required here.

4.4 Centrality

4.4.1 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.

networkx.algorithms.centrality.degree_centrality

networkx.algorithms.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.

networkx.algorithms.centrality.in_degree_centrality

networkx.algorithms.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.

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.

networkx.algorithms.centrality.out_degree_centrality

networkx.algorithms.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.

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.

4.4.2 Closeness

closeness_centrality(G[, v, distance, ...])	Compute closeness centrality for nodes.

networkx.algorithms.centrality.closeness_centrality

networkx.algorithms.centrality.**closeness_centrality**(*G*, *v=None*, *distance=None*, *normalized=True*)

Compute closeness centrality for nodes.

Closeness centrality at a node is 1/average distance to all other nodes.

Parameters **G** : graph

A networkx graph

v : node, optional

Return only the value for node v

distance : string key, optional (default=None)

Use specified edge key as edge distance. If True, use 'weight' as the edge key.

normalized : bool, optional

If True (default) normalize by the graph size.

Returns **nodes** : dictionary

Dictionary of nodes with closeness centrality as the value.

See Also:

betweenness_centrality, load_centrality, eigenvector_centrality, degree_centrality

Notes

The closeness centrality is normalized to 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.

4.4.3 Betweenness

betweenness_centrality(G[, normalized, ...])	Compute the shortest-path betweenness centrality for nodes.
edge_betweenness_centrality(G[, normalized, ...])	Compute betweenness centrality for edges.

networkx.algorithms.centrality.betweenness_centrality

networkx.algorithms.centrality.**betweenness_centrality**(*G*, *normalized=True*, *weight=None*, *endpoints=False*)

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 $v \in s,t$, $\sigma(s,t|v)=0$ [R87].

Parameters **G** : graph

A NetworkX graph

normalized : bool, optional

If True the betweenness values are normalized by $1/(n-1)(n-2)$ where n is the number of nodes in G.

weight : None, True or string, optional

If None, all edge weights are considered equal. If True, edge attribute 'weight' is used as weight of each edge. Otherwise holds the name of the edge attribute used as weight.

endpoints : bool, optional

If True include the endpoints in the shortest path counts.

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 [R86]. See [R87] for details on algorithms for variations and related metrics.

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

[R86], [R87]

networkx.algorithms.centrality.edge_betweenness_centrality

networkx.algorithms.centrality.**edge_betweenness_centrality**(*G*, *normalized=True*, *weight=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:

$$c_B(v) = \sum_{s,t \in V} \frac{\sigma(s,t|e)}{\sigma(s,t)}$$

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 [R93].

Parameters **G** : graph

A NetworkX graph

normalized : bool, optional

If True the betweenness values are normalized by $1/(n-1)(n-2)$ where n is the number of nodes in G.

weight : None, True or string, optional

If None, all edge weights are considered equal. If True, edge attribute 'weight' is used as weight of each edge. Otherwise holds the name of the edge attribute used as weight.

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 [R92].

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

[R92], [R93]

4.4.4 Current Flow Closeness

current_flow_closeness_centrality(G[, ...])	Compute current-flow closeness centrality for nodes.

networkx.algorithms.centrality.current_flow_closeness_centrality

networkx.algorithms.centrality.**current_flow_closeness_centrality**(*G*, *normalized=True*)

Compute current-flow closeness centrality for nodes.

A 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

normalized : bool, optional

If True the values are normalized by 1/(n-1) where n is the number of nodes in G.

Returns **nodes** : dictionary

Dictionary of nodes with current flow closeness centrality as the value.

See Also:

closeness_centrality

Notes

The algorithm is from Brandes [R90].

See also [R91] for the original definition of information centrality.

References

[R90], [R91]

4.4.5 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.

networkx.algorithms.centrality.current_flow_betweenness_centrality

networkx.algorithms.centrality.**current_flow_betweenness_centrality**(*G*, *normalized=True*, *weight='weight'*)

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 [R89].

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 : string or None, optional (default='weight')

Key for edge data used as the edge weight. If None, then use 1 as each edge weight.

Returns **nodes** : dictionary

Dictionary of nodes with betweenness centrality as the value.

See Also:

betweenness_centrality, edge_betweenness_centrality, edge_current_flow_betweenness_centra

Notes

The algorithm is from Brandes [R88].

If the edges have a 'weight' attribute they will be used as weights in this algorithm. Unspecified weights are set to 1.

References

[R88], [R89]

networkx.algorithms.centrality.edge_current_flow_betweenness_centrality

networkx.algorithms.centrality.**edge_current_flow_betweenness_centrality**(*G*, *normalized=True*, *weight='weight'*)

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 [R95].

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 : string or None, optional (default='weight')

Key for edge data used as the edge weight. If None, then use 1 as each edge weight.

Returns **nodes** : dictionary

Dictionary of edge tuples with betweenness centrality as the value.

See Also:

betweenness_centrality, edge_betweenness_centrality, current_flow_betweenness_centrality

Notes

The algorithm is from Brandes [R94].

If the edges have a 'weight' attribute they will be used as weights in this algorithm. Unspecified weights are set to 1.

References

[R94], [R95]

4.4.6 Eigenvector

eigenvector_centrality(G[, max_iter, tol, ...])	Compute the eigenvector centrality for the graph G.
eigenvector_centrality_numpy(G)	Compute the eigenvector centrality for the graph G.

networkx.algorithms.centrality.eigenvector_centrality

networkx.algorithms.centrality.**eigenvector_centrality**(*G*, *max_iter=100*, *tol=9.9999999999999995e-07*, *nstart=None*)

Compute the eigenvector centrality for the graph G.

Uses the power method to find the eigenvector for the largest eigenvalue of the adjacency matrix of G.

Parameters **G** : graph

A networkx graph

max_iter : interger, 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 eigenvector iteration for each node.

Returns **nodes** : dictionary

Dictionary of nodes with eigenvector centrality as the value.

See Also:

`eigenvector_centrality_numpy`, `pagerank`, `hits`

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.

For directed graphs this is "right" eigevector centrality. For "left" eigenvector centrality, first reverse the graph with G.reverse().

Examples

```
>>> G=nx.path_graph(4)
>>> centrality=nx.eigenvector_centrality(G)
>>> print(['%s %0.2f'%(node,centrality[node]) for node in centrality])
['0 0.37', '1 0.60', '2 0.60', '3 0.37']
```

networkx.algorithms.centrality.eigenvector_centrality_numpy

networkx.algorithms.centrality.**eigenvector_centrality_numpy**(*G*)

Compute the eigenvector centrality for the graph G.

Parameters **G** : graph

A networkx graph

Returns **nodes** : dictionary

Dictionary of nodes with eigenvector centrality as the value.

See Also:

`eigenvector_centrality`, `pagerank`, `hits`

Notes

This algorithm uses the NumPy eigenvalue solver.

For directed graphs this is “right” eigevector centrality. For “left” eigenvector centrality, first reverse the graph with G.reverse().

Examples

```
>>> G=nx.path_graph(4)
>>> centrality=nx.eigenvector_centrality_numpy(G)
>>> print(['%s %0.2f'%(node,centrality[node]) for node in centrality])
['0 0.37', '1 0.60', '2 0.60', '3 0.37']
```

4.4.7 Load

load_centrality(G[, v, cutoff, normalized, ...])	Compute load centrality for nodes.
edge_load(G[, nodes, cutoff])	Compute edge load.

networkx.algorithms.centrality.load_centrality

networkx.algorithms.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

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, True or string, optional

If None, edge weights are ignored. If True, edge attribute ‘weight’ is used as weight of each edge. Otherwise holds the name of the edge attribute used as weight.

cutoff : bool, optional

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. For this load algorithm see the reference Scientific collaboration networks: II. Shortest paths, weighted networks, and centrality, M. E. J. Newman, Phys. Rev. E 64, 016132 (2001).

networkx.algorithms.centrality.edge_load

networkx.algorithms.centrality.**edge_load**(*G*, *nodes=None*, *cutoff=False*)

Compute edge load.

WARNING:

This module is for demonstration and testing purposes.

4.5 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). http://en.wikipedia.org/wiki/Chordal_graph

is_chordal(G)	Checks whether G is a chordal graph.
chordal_graph_cliques(G)	Returns the set of maximal cliques of a chordal graph.
chordal_graph_treewidth(G)	Returns the treewidth of the chordal graph G.
find_induced_nodes(G, s, t[, treewidth_bound])	Returns the set of induced nodes in the path from s to t.

4.5.1 networkx.algorithms.chordal.chordal_alg.is_chordal

networkx.algorithms.chordal.chordal_alg.**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 **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.

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 [R98].

References

[R98]

Examples

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

4.5.2 networkx.algorithms.chordal.chordal_alg.chordal_graph_cliques

networkx.algorithms.chordal.chordal_alg.**chordal_graph_cliques**(*G*)

Returns the set of 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

Returns **cliques** : A set containing the maximal cliques 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.

Examples

```
>>> import networkx as nx
>>> 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)
>>> setlist = nx.chordal_graph_cliques(G)
```

4.5.3 networkx.algorithms.chordal.chordal_alg.chordal_graph_treewidth

networkx.algorithms.chordal.chordal_alg.**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. 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.

References

[R96]

Examples

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

4.5.4 networkx.algorithms.chordal.chordal_alg.find_induced_nodes

networkx.algorithms.chordal.chordal_alg.**find_induced_nodes**(*G*, *s*, *t*, *treewidth_bound=2147483647*)

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

treewith_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 **I** : 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 [R97]. A formal definition of induced node can also be found on that reference.

References

[R97]

Examples

```
>>> import networkx as nx
>>> G=nx.Graph()
>>> G = nx.generators.classic.path_graph(10)
>>> I = nx.find_induced_nodes(G,1,9,2)
>>> list(I)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

4.6 Clique

Find and manipulate cliques of graphs.

Note that finding the largest clique of a graph has been shown to be an NP-complete problem; the algorithms here could take a long time to run.

http://en.wikipedia.org/wiki/Clique_problem

find_cliques(G)	Search for all maximal cliques in a graph.
make_max_clique_graph(G[, create_using, name])	Create the maximal clique graph of a graph.
make_clique_bipartite(G[, fpos, ...])	Create a bipartite clique graph from a graph G.
graph_clique_number(G[, cliques])	Return the clique number (size of the largest clique) for G.
graph_number_of_cliques(G[, cliques])	Returns the number of maximal cliques in G.
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.

4.6.1 networkx.algorithms.clique.find_cliques

networkx.algorithms.clique.**find_cliques**(*G*)

Search for all maximal cliques in a graph.

This algorithm searches for maximal cliques in a graph. maximal cliques are the largest complete subgraph containing a given point. The largest maximal clique is sometimes called the maximum clique.

This implementation is a generator of lists each of which contains the members of a maximal clique. To obtain a list of cliques, use list(find_cliques(G)). The method essentially unrolls the recursion used in the references to avoid issues of recursion stack depth.

See Also:

find_cliques_recursive, A

Notes

Based on the algorithm published by Bron & Kerbosch (1973) [R99] as adapated by Tomita, Tanaka and Takahashi (2006) [R100] and discussed in Cazals and Karande (2008) [R101].

There are often many cliques in graphs. This algorithm can run out of memory for large graphs.

References

[R99], [R100], [R101]

4.6.2 networkx.algorithms.clique.make_max_clique_graph

networkx.algorithms.clique.**make_max_clique_graph**(*G*, *create_using=None*, *name=None*)

Create the maximal clique graph of a graph.

Finds the maximal cliques and treats these as nodes. The nodes are connected if they have common members in the original graph. Theory has done a lot with clique graphs, but I haven't seen much on maximal clique graphs.

Notes

This should be the same as make_clique_bipartite followed by project_up, but it saves all the intermediate steps.

4.6.3 networkx.algorithms.clique.make_clique_bipartite

networkx.algorithms.clique.**make_clique_bipartite**(*G*, *fpos=None*, *create_using=None*, *name=None*)

Create a bipartite clique graph from a graph G.

Nodes of G are retained as the "bottom nodes" of B and cliques of G become "top nodes" of B. Edges are present if a bottom node belongs to the clique represented by the top node.

Returns a Graph with additional attribute dict B.node_type which is keyed by nodes to "Bottom" or "Top" appropriately.

if fpos is not None, a second additional attribute dict B.pos is created to hold the position tuple of each node for viewing the bipartite graph.

4.6.4 networkx.algorithms.clique.graph_clique_number

networkx.algorithms.clique.**graph_clique_number**(*G*, *cliques=None*)

Return the clique number (size of the largest clique) for G.

An optional list of cliques can be input if already computed.

4.6.5 networkx.algorithms.clique.graph_number_of_cliques

networkx.algorithms.clique.**graph_number_of_cliques**(*G*, *cliques=None*)
Returns the number of maximal cliques in G.

An optional list of cliques can be input if already computed.

4.6.6 networkx.algorithms.clique.node_clique_number

networkx.algorithms.clique.**node_clique_number**(*G*, *nodes=None*, *cliques=None*)
Returns the size of the largest maximal clique containing each given node.

Returns a single or list depending on input nodes. Optional list of cliques can be input if already computed.

4.6.7 networkx.algorithms.clique.number_of_cliques

networkx.algorithms.clique.**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.

4.6.8 networkx.algorithms.clique.cliques_containing_node

networkx.algorithms.clique.**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.

4.7 Clustering

Algorithms to characterize the number of triangles in a graph.

triangles(G[, nodes])	Compute the number of triangles.
transitivity(G)	Compute transitivity.
clustering(G[, nodes, weighted])	Compute the clustering coefficient for nodes.
average_clustering(G[, weighted])	Compute average clustering coefficient.
square_clustering(G[, nodes])	Compute the squares clustering coefficient for nodes.

4.7.1 networkx.algorithms.cluster.triangles

networkx.algorithms.cluster.**triangles**(*G*, *nodes=None*)
Compute the number of triangles.

Finds the number of triangles that include a node as one of the vertices.

Parameters **G** : graph

A networkx graph

nodes : container of nodes, optional

Compute triangles for nodes. The default is all nodes in G.

Returns **out** : dictionary

Number of trianges 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]
```

4.7.2 networkx.algorithms.cluster.transitivity

networkx.algorithms.cluster.**transitivity**(*G*)

Compute transitivity.

Finds the fraction of all possible triangles which are in fact triangles. Possible triangles are identified by the number of "triads" (two edges with a shared vertex).

T = 3*triangles/triads

Parameters **G** : graph

A networkx graph

Returns **out** : float

Transitivity

Examples

```
>>> G=nx.complete_graph(5)
>>> print(nx.transitivity(G))
1.0
```

4.7.3 networkx.algorithms.cluster.clustering

networkx.algorithms.cluster.**clustering**(*G*, *nodes=None*, *weighted=False*)

Compute the clustering coefficient for nodes.

For each node find the fraction of possible triangles that exist,

$$c_v = \frac{2T(v)}{deg(v)(deg(v)-1)}$$

where $T(v)$ is the number of triangles through node v.

Parameters **G** : graph

A networkx graph

nodes : container of nodes, optional

Limit to specified nodes. Default is entire graph.

weighted : bool, optional

If True use weights on edges in computing clustering coefficients.

Returns **out** : float, dictionary or tuple of dictionaries

Clustering coefficient at specified nodes

Notes

Self loops are ignored.

References

[R103]

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}
```

4.7.4 networkx.algorithms.cluster.average_clustering

networkx.algorithms.cluster.**average_clustering**(*G*, *weighted=False*)

Compute average 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.

Parameters **G** : graph

A networkx graph

weighted : bool, optional

If True use weights on edges in computing clustering coefficients.

Returns **out** : float

Average clustering

Notes

This is a space saving routine; it might be faster to use clustering to get a list and then take the average.

Self loops are ignored.

References

[R102]

Examples

```
>>> G=nx.complete_graph(5)
>>> print(nx.average_clustering(G))
1.0
```

4.7.5 networkx.algorithms.cluster.square_clustering

networkx.algorithms.cluster.**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 [R104]

$$C_4(v) = \frac{\sum_{u=1}^{k_v} \sum_{w=u+1}^{k_v} q_v(u,w)}{\sum_{u=1}^{k_v} \sum_{w=u+1}^{k_v} [a_v(u,w) + q_v(u,w)]}$$

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.

Parameters **G** : graph

A NetworkX graph

nodes : container of nodes, optional

Compute clustering only for specified nodes. Default is entire graph.

Returns **c4** : dictionary

A dictionary keyed by node with the square clustering coefficient value.

Notes

While $C_3(v)$ 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

[R104]

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}
```

4.8 Components

4.8.1 Connectivity

Connected components.

is_connected(G)	Test graph connectivity.
number_connected_components(G)	Return number of connected components in graph.
connected_components(G)	Return nodes in connected components of graph.
connected_component_subgraphs(G)	Return connected components as subgraphs.
node_connected_component(G, n)	Return nodes in connected components of graph containing node n.

networkx.algorithms.components.connected.is_connected

networkx.algorithms.components.connected.**is_connected**(*G*)

Test graph connectivity.

Parameters **G** : NetworkX Graph

An undirected graph.

Returns **connected** : bool

True if the graph is connected, false otherwise.

See Also:

connected_components

Notes

For undirected graphs only.

Examples

```
>>> G=nx.path_graph(4)
>>> print(nx.is_connected(G))
True
```

networkx.algorithms.components.connected.number_connected_components

networkx.algorithms.components.connected.**number_connected_components**(*G*)

Return number of connected components in graph.

Parameters **G** : NetworkX Graph

An undirected graph.

Returns **n** : integer

Number of connected components

See Also:

`connected_components`

Notes

For undirected graphs only.

networkx.algorithms.components.connected.connected_components

`networkx.algorithms.components.connected.`**`connected_components`**`(`G`)`

Return nodes in connected components of graph.

Parameters **G** : NetworkX Graph

An undirected graph.

Returns **comp** : list of lists

A list of nodes for each component of G.

See Also:

`strongly_connected_components`

Notes

The list is ordered from largest connected component to smallest. For undirected graphs only.

networkx.algorithms.components.connected.connected_component_subgraphs

`networkx.algorithms.components.connected.`**`connected_component_subgraphs`**`(`G`)`

Return connected components as subgraphs.

Parameters **G** : NetworkX Graph

An undirected graph.

Returns **glist** : list

A list of graphs, one for each connected component of G.

See Also:

`connected_components`

Notes

The list is ordered from largest connected component to smallest. For undirected graphs only.

Examples

Get largest connected component as subgraph

```
>>> G=nx.path_graph(4)
>>> G.add_edge(5,6)
>>> H=nx.connected_component_subgraphs(G)[0]
```

networkx.algorithms.components.connected.node_connected_component

networkx.algorithms.components.connected.**node_connected_component**(*G*, *n*)

Return nodes in connected components of graph containing node n.

Parameters **G** : NetworkX Graph

An undirected graph.

n : node label

A node in G

Returns **comp** : lists

A list of nodes in component of G containing node n.

See Also:

connected_components

Notes

For undirected graphs only.

4.8.2 Strong connectivity

Strongly connected components.

is_strongly_connected(G)	Test directed graph for strong connectivity.
number_strongly_connected_components(G)	Return number of strongly connected components in graph.
strongly_connected_components(G)	Return nodes in strongly connected components of graph.
strongly_connected_component_subgraphs(G)	Return strongly connected components as subgraphs.
strongly_connected_components_recursive(G)	Return nodes in strongly connected components of graph.
kosaraju_strongly_connected_components(G[, ...])	Return nodes in strongly connected components of graph.
condensation(G)	Returns the condensation of G.

networkx.algorithms.components.strongly_connected.is_strongly_connected

networkx.algorithms.components.strongly_connected.**is_strongly_connected**(*G*)

Test directed graph for strong connectivity.

Parameters **G** : NetworkX Graph

A directed graph.

Returns **connected** : bool

True if the graph is strongly connected, False otherwise.

See Also:

`strongly_connected_components`

Notes

For directed graphs only.

networkx.algorithms.components.strongly_connected.number_strongly_connected_components

`networkx.algorithms.components.strongly_connected.`**`number_strongly_connected_components`**`(`G`)`

Return number of strongly connected components in graph.

Parameters **G** : NetworkX graph

A directed graph.

Returns **n** : integer

Number of strongly connected components

See Also:

`connected_components`

Notes

For directed graphs only.

networkx.algorithms.components.strongly_connected.strongly_connected_components

`networkx.algorithms.components.strongly_connected.`**`strongly_connected_components`**`(`G`)`

Return nodes in strongly connected components of graph.

Parameters **G** : NetworkX Graph

An directed graph.

Returns **comp** : list of lists

A list of nodes for each component of G. The list is ordered from largest connected component to smallest.

See Also:

`connected_components`

Notes

Uses Tarjan's algorithm with Nuutila's modifications. Nonrecursive version of algorithm.

References

[R105], [R106]

networkx.algorithms.components.strongly_connected.strongly_connected_component_subgraphs

networkx.algorithms.components.strongly_connected.**strongly_connected_component_subgraphs**(*G*

Return strongly connected components as subgraphs.

Parameters **G** : NetworkX Graph

A graph.

Returns **glist** : list

A list of graphs, one for each strongly connected component of G.

See Also:

`connected_component_subgraphs`

Notes

The list is ordered from largest strongly connected component to smallest.

networkx.algorithms.components.strongly_connected.strongly_connected_components_recursive

networkx.algorithms.components.strongly_connected.**strongly_connected_components_recursive**(

Return nodes in strongly connected components of graph.

Recursive version of algorithm.

Parameters **G** : NetworkX Graph

An directed graph.

Returns **comp** : list of lists

A list of nodes for each component of G. The list is ordered from largest connected component to smallest.

See Also:

`connected_components`

Notes

Uses Tarjan's algorithm with Nuutila's modifications.

References

[R107], [R108]

networkx.algorithms.components.strongly_connected.kosaraju_strongly_connected_components

networkx.algorithms.components.strongly_connected.**kosaraju_strongly_connected_components**(*G*, *so*

Return nodes in strongly connected components of graph.

Parameters **G** : NetworkX Graph

An directed graph.

Returns **comp** : list of lists

A list of nodes for each component of G. The list is ordered from largest connected component to smallest.

See Also:

connected_components

Notes

Uses Kosaraju's algorithm.

networkx.algorithms.components.strongly_connected.condensation

networkx.algorithms.components.strongly_connected.**condensation**(*G*)

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 Graph

A directed graph.

Returns **cG** : NetworkX DiGraph

The condensation of G.

Notes

After contracting all strongly connected components to a single node, the resulting graph is a directed acyclic graph.

4.8.3 Weak connectivity

Weakly connected components.

is_weakly_connected(G)	Test directed graph for weak connectivity.
number_weakly_connected_components(G)	Return the number of connected components in G.
weakly_connected_components(G)	Return weakly connected components of G.
weakly_connected_component_subgraphs(G)	Return weakly connected components as subgraphs.

networkx.algorithms.components.weakly_connected.is_weakly_connected

networkx.algorithms.components.weakly_connected.**is_weakly_connected**(*G*)
Test directed graph for weak connectivity.

Parameters **G** : NetworkX Graph

A directed graph.

Returns **connected** : bool

True if the graph is weakly connected, False otherwise.

See Also:

strongly_connected_components

Notes

For directed graphs only.

networkx.algorithms.components.weakly_connected.number_weakly_connected_components

networkx.algorithms.components.weakly_connected.**number_weakly_connected_components**(*G*)
Return the number of connected components in G. For directed graphs only.

networkx.algorithms.components.weakly_connected.weakly_connected_components

networkx.algorithms.components.weakly_connected.**weakly_connected_components**(*G*)
Return weakly connected components of G.

networkx.algorithms.components.weakly_connected.weakly_connected_component_subgraphs

networkx.algorithms.components.weakly_connected.**weakly_connected_component_subgraphs**(*G*)
Return weakly connected components as subgraphs.

4.8.4 Atrracting components

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)	Returns a list of attracting components in G.
attracting_component_subgraphs(G)	Returns a list of attracting component subgraphs from G.

networkx.algorithms.components.attracting.is_attracting_component

networkx.algorithms.components.attracting.**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.

See Also:

attracting_components, number_attracting_components, attracting_component_subgraphs

networkx.algorithms.components.attracting.number_attracting_components

networkx.algorithms.components.attracting.**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.

See Also:

attracting_components, is_attracting_component, attracting_component_subgraphs

networkx.algorithms.components.attracting.attracting_components

networkx.algorithms.components.attracting.**attracting_components**(G)

Returns a list of 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.

Parameters **G** : DiGraph, MultiDiGraph

The graph to be analyzed.

Returns **attractors** : list

The list of attracting components, sorted from largest attracting component to smallest attracting component.

See Also:

number_attracting_components, is_attracting_component, attracting_component_subgraphs

networkx.algorithms.components.attracting.attracting_component_subgraphs

networkx.algorithms.components.attracting.**attracting_component_subgraphs**(G)

Returns a list of attracting component subgraphs from G.

Parameters **G** : DiGraph, MultiDiGraph

The graph to be analyzed.

Returns **subgraphs** : list

A list of node-induced subgraphs of the attracting components of G.

See Also:

attracting_components, number_attracting_components, is_attracting_component

4.9 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 reference for details:

An O(m) Algorithm for Cores Decomposition of Networks Vladimir Batagelj and Matjaz Zaversnik, 2003. http://arxiv.org/abs/cs.DS/0310049

core_number(G)	Return the core number for each vertex.
k_core(G[, k, core_number])	Return the k-core of G.
k_shell(G[, k, core_number])	Return the k-shell of G.
k_crust(G[, k, core_number])	Return the k-crust of G.
k_corona(G, k[, core_number])	Return the k-crust of G.

4.9.1 networkx.algorithms.core.core_number

networkx.algorithms.core.**core_number**(*G*)

Return 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 defined 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

[R109]

4.9.2 networkx.algorithms.core.k_core

networkx.algorithms.core.**k_core**(*G*, *k=None*, *core_number=None*)

Return 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.

References

[R110]

4.9.3 networkx.algorithms.core.k_shell

networkx.algorithms.core.**k_shell**(*G*, *k=None*, *core_number=None*)

Return the k-shell of G.

The k-shell is the subgraph of nodes in the k-core containing nodes of exactly degree k.

Parameters **G** : NetworkX graph

A graph or directed graph.

k : int, optional

The order of the shell. If not specified return the main 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 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.

References

[R113]

4.9.4 networkx.algorithms.core.k_crust

`networkx.algorithms.core.`**`k_crust`**(*G*, *k=None*, *core_number=None*)

Return the k-crust of G.

The k-crust is the graph G with the k-core 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 defined for graphs with self loops or parallel edges.

See Also:

`core_number`

Notes

This definition of k-crust is different than the definition in [R112]. The k-crust in [R112] 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.

References

[R112]

4.9.5 networkx.algorithms.core.k_corona

networkx.algorithms.core.**k_corona**(*G*, *k*, *core_number=None*)

Return the k-crust of G.

The k-corona is the subset of vertices 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.

References

[R111]

4.10 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.

4.10.1 networkx.algorithms.cycles.cycle_basis

networkx.algorithms.cycles.**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 [R114].

References

[R114]

Examples

```
>>> G=nx.Graph()
>>> G.add_cycle([0,1,2,3])
>>> G.add_cycle([0,3,4,5])
>>> print(nx.cycle_basis(G,0))
[[3, 4, 5, 0], [1, 2, 3, 0]]
```

4.10.2 networkx.algorithms.cycles.simple_cycles

networkx.algorithms.cycles.**simple_cycles**(*G*)

Find simple cycles (elementary circuits) of a directed graph.

An simple cycle, or elementary circuit, is a closed path where no node appears twice, except that the first and last node are the same. Two elementary circuits are distinct if they are not cyclic permutations of each other.

Parameters **G** : NetworkX DiGraph

A directed graph

Returns **A list of circuits, where each circuit is a list of nodes, with the first** :

and last node being the same. :

Example: :

>>> G = nx.DiGraph([(0, 0), (0, 1), (0, 2), (1, 2), (2, 0), (2, 1), (2, 2)]) :

>>> nx.simple_cycles(G) :

[[0, 0], [0, 1, 2, 0], [0, 2, 0], [1, 2, 1], [2, 2]] :

See Also:

cycle_basis

Notes

The implementation follows pp. 79-80 in [R115].

The time complexity is O((n+e)(c+1)) for n nodes, e edges and c elementary circuits.

References

[R115]

4.11 Directed Acyclic Graphs

Algorithms for directed acyclic graphs (DAGs).

topological_sort(G[, nbunch])	Return a list of nodes in topological sort order.
topological_sort_recursive(G[, nbunch])	Return a list of nodes in topological sort order.
is_directed_acyclic_graph(G)	Return True if the graph G is a directed acyclic graph (DAG) or

4.11.1 networkx.algorithms.dag.topological_sort

networkx.algorithms.dag.**topological_sort**(*G*, *nbunch=None*)

Return a list of nodes in topological sort order.

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

nbunch : container of nodes (optional)

Explore graph in specified order given in nbunch

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 NetworkX-Unfeasible exception is raised.

See Also:

is_directed_acyclic_graph

Notes

This algorithm is based on a description and proof in The Algorithm Design Manual [R116] .

References

[R116]

4.11.2 networkx.algorithms.dag.topological_sort_recursive

networkx.algorithms.dag.**topological_sort_recursive**(*G*, *nbunch=None*)

Return a list of nodes in topological sort order.

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

nbunch : container of nodes (optional)

Explore graph in specified order given in nbunch

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 NetworkX-Unfeasible exception is raised.

See Also:

topological_sort, is_directed_acyclic_graph

Notes

This is a recursive version of topological sort.

4.11.3 networkx.algorithms.dag.is_directed_acyclic_graph

networkx.algorithms.dag.**is_directed_acyclic_graph**(*G*)

Return True if the graph G is a directed acyclic graph (DAG) or False if not.

Parameters **G** : NetworkX graph

A graph

Returns **is_dag** : bool

True if G is a DAG, false otherwise

4.12 Distance Measures

Graph diameter, radius, eccentricity and other properties.

center(G[, e])	Return the periphery of the graph G.
diameter(G[, e])	Return the diameter of the graph G.
eccentricity(G[, v, sp])	Return the eccentricity of nodes in G.
periphery(G[, e])	Return the periphery of the graph G.
radius(G[, e])	Return the radius of the graph G.

4.12.1 networkx.algorithms.distance_measures.center

networkx.algorithms.distance_measures.**center**(*G*, *e=None*)

Return the periphery 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.

Returns **c** : list

List of nodes in center

4.12.2 networkx.algorithms.distance_measures.diameter

networkx.algorithms.distance_measures.**diameter**(*G*, *e=None*)

Return 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.

Returns **d** : integer

Diameter of graph

See Also:

eccentricity

4.12.3 networkx.algorithms.distance_measures.eccentricity

networkx.algorithms.distance_measures.**eccentricity**(*G*, *v=None*, *sp=None*)

Return 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 lenghts as a dictionary of dictionaries

Returns **ecc** : dictionary

A dictionary of eccentricity values keyed by node.

4.12.4 networkx.algorithms.distance_measures.periphery

networkx.algorithms.distance_measures.**periphery**(*G*, *e=None*)

Return 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.

Returns **p** : list

List of nodes in periphery

4.12.5 networkx.algorithms.distance_measures.radius

networkx.algorithms.distance_measures.**radius**(*G*, *e=None*)

Return 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.

Returns **r** : integer

Radius of graph

4.13 Distance-Regular Graphs

is_distance_regular(G)	Returns True if the graph is distance regular, False otherwise.
intersection_array(G)	Returns the intersection array of a distance-regular graph.
global_parameters(b, c)	Return global parameters for a given intersection array.

4.13.1 networkx.algorithms.distance_regular.is_distance_regular

networkx.algorithms.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

[R119], [R120]

Examples

```
>>> G=nx.hypercube_graph(6)
>>> nx.is_distance_regular(G)
True
```

4.13.2 networkx.algorithms.distance_regular.intersection_array

networkx.algorithms.distance_regular.**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'sintersection 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

[R118]

Examples

```
>>> G=nx.icosahedral_graph()
>>> nx.intersection_array(G)
([5, 2, 1], [1, 2, 5])
```

4.13.3 networkx.algorithms.distance_regular.global_parameters

networkx.algorithms.distance_regular.**global_parameters**(*b*, *c*)

Return 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,c: tuple of lists** :

Returns **p** : list of three-tuples

See Also:

intersection_array

References

[R117]

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)]
```

4.14 Eulerian

Eulerian circuits and graphs.

is_eulerian(G)	Return True if G is an Eulerian graph, False otherwise.
eulerian_circuit(G[, source])	Return the edges of an Eulerian circuit in G.

4.14.1 networkx.algorithms.euler.is_eulerian

networkx.algorithms.euler.**is_eulerian**(*G*)

Return True if G is an Eulerian graph, False otherwise.

An Eulerian graph is a graph with an Eulerian circuit.

Parameters **G** : graph

A NetworkX Graph

Notes

This implementation requires the graph to be connected (or strongly connected for directed graphs).

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
```

4.14.2 networkx.algorithms.euler.eulerian_circuit

networkx.algorithms.euler.**eulerian_circuit**(*G*, *source=None*)

Return the edges of an Eulerian circuit in G.

An Eulerian circuit is a path that crosses every edge in G exactly once and finishes at the starting node.

Parameters **G** : graph

A NetworkX Graph

source : node, optional

Starting node for circuit.

Returns **edges** : generator

A generator that produces edges in the Eulerian circuit.

Raises **NetworkXError** :

If the graph is not Eulerian.

See Also:

is_eulerian

Notes

Uses Fleury's algorithm [R121],[R122]_

References

[R121], [R122]

Examples

```
>>> G=nx.complete_graph(3)
>>> list(nx.eulerian_circuit(G))
[(0, 1), (1, 2), (2, 0)]
>>> list(nx.eulerian_circuit(G,source=1))
[(1, 0), (0, 2), (2, 1)]
>>> [u for u,v in nx.eulerian_circuit(G)]  # nodes in circuit
[0, 1, 2]
```

4.15 Flows

4.15.1 Ford-Fulkerson

max_flow(G, s, t[, capacity])	Find the value of a maximum single-commodity flow.
min_cut(G, s, t[, capacity])	Compute the value of a minimum (s, t)-cut.
ford_fulkerson(G, s, t[, capacity])	Find a maximum single-commodity flow using the Ford-Fulkerson
ford_fulkerson_flow(G, s, t[, capacity])	Return a maximum flow for a single-commodity flow problem.

networkx.algorithms.flow.max_flow

networkx.algorithms.flow.**max_flow**(*G*, *s*, *t*, *capacity='capacity'*)

Find the value of a maximum single-commodity flow.

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'.

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.

Examples

```
>>> import networkx as nx
>>> 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)
>>> flow = nx.max_flow(G, 'x', 'y')
>>> flow
3.0
```

networkx.algorithms.flow.min_cut

networkx.algorithms.flow.**min_cut**(*G*, *s*, *t*, *capacity='capacity'*)

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 **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'.

Returns **cutValue** : 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.

Examples

```
>>> import networkx as nx
>>> 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)
>>> nx.min_cut(G, 'x', 'y')
3.0
```

networkx.algorithms.flow.ford_fulkerson

networkx.algorithms.flow.**ford_fulkerson**(*G*, *s*, *t*, *capacity='capacity'*)

Find a maximum single-commodity flow using the Ford-Fulkerson algorithm.

This algorithm uses Edmonds-Karp-Dinitz path selection rule which guarantees 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'.

Returns **flow_value** : integer, float

Value of the maximum flow, i.e., net outflow from the source.

flow_dict : dictionary

Dictionary of dictionaries keyed by nodes such that flow_dict[u][v] is the flow edge (u, v).

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.

Examples

```
>>> import networkx as nx
>>> 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)
>>> flow, F = nx.ford_fulkerson(G, 'x', 'y')
>>> flow
3.0
```

networkx.algorithms.flow.ford_fulkerson_flow

networkx.algorithms.flow.**ford_fulkerson_flow**(*G*, *s*, *t*, *capacity='capacity'*)

Return a maximum flow for a single-commodity flow problem.

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'.

Returns **flow_dict** : dictionary

Dictionary of dictionaries keyed by nodes such that flow_dict[u][v] is the flow edge (u, v).

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.

Examples

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

```
>>> F = nx.ford_fulkerson_flow(G, 'x', 'y')
>>> for u, v in G.edges_iter():
...     print('(%s, %s) %.2f' % (u, v, F[u][v]))
...
(a, c) 2.00
(c, y) 2.00
(b, c) 0.00
(b, d) 1.00
(e, y) 1.00
(d, e) 1.00
(x, a) 2.00
(x, b) 1.00
```

4.15.2 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])	Return 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])	Return a maximum (s, t)-flow of minimum cost.

networkx.algorithms.flow.network_simplex

networkx.algorithms.flow.**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

References

W. J. Cook, W. H. Cunningham, W. R. Pulleyblank and A. Schrijver. Combinatorial Optimization. Wiley-Interscience, 1998.

Examples

A simple example of a min cost flow problem.

```
>>> import networkx as nx
>>> 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': {'c': 1, 'b': 4}, 'c': {'d': 1}, 'b': {'d': 4}, 'd': {}}
```

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 = True)
True
>>> [(u, v) for u in flowDict for v in flowDict[u] if flowDict[u][v] > 0]
[('x', 'u'), ('s', 'x'), ('u', 'v')]
>>> nx.shortest_path(G, 's', 'v', weight = True)
['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
{'a': {'t': 4}, 'd': {'w': 2}, 'q': {'d': 1}, 'p': {'q': 2, 'a': 2}, 't': {'q': 1, 'w': 1}, 'w
```

networkx.algorithms.flow.min_cost_flow_cost

networkx.algorithms.flow.**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`

Examples

A simple example of a min cost flow problem.

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

networkx.algorithms.flow.min_cost_flow

networkx.algorithms.flow.**min_cost_flow**(*G*, *demand='demand'*, *capacity='capacity'*, *weight='weight'*)

Return 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`

Examples

A simple example of a min cost flow problem.

```
>>> import networkx as nx
>>> 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)
>>> flowDict
{'a': {'c': 1, 'b': 4}, 'c': {'d': 1}, 'b': {'d': 4}, 'd': {}}
```

networkx.algorithms.flow.cost_of_flow

networkx.algorithms.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

networkx.algorithms.flow.max_flow_min_cost

networkx.algorithms.flow.**max_flow_min_cost**(*G*, *s*, *t*, *capacity='capacity'*, *weight='weight'*)

Return 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, ford_fulkerson, min_cost_flow, min_cost_flow_cost, network_simplex

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)
>>> nx.cost_of_flow(G, mincostFlow)
373
>>> maxFlow = nx.ford_fulkerson_flow(G, 1, 7)
>>> nx.cost_of_flow(G, maxFlow)
428
>>> mincostFlowValue = (sum((mincostFlow[u][7] for u in G.predecessors(7)))
...                     - sum((mincostFlow[7][v] for v in G.successors(7))))
>>> mincostFlowValue == nx.max_flow(G, 1, 7)
True
```

4.16 Isolates

Functions for identifying isolate (degree zero) nodes.

is_isolate(G, n)	Determine of node n is an isolate (degree zero).
isolates(G)	Return list of isolates in the graph.

4.16.1 networkx.algorithms.isolate.is_isolate

networkx.algorithms.isolate.**is_isolate**(*G*, *n*)

Determine of node n is an isolate (degree zero).

Parameters **G** : graph

A networkx graph

n : node

A node in G

Returns **isolate** : bool

True if n has no neighbors, False otherwise.

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
```

4.16.2 networkx.algorithms.isolate.isolates

networkx.algorithms.isolate.**isolates**(*G*)

Return list of isolates in the graph.

Isolates are nodes with no neighbors (degree zero).

Parameters **G** : graph

A networkx graph

Returns **isolates** : list

List of isolate nodes.

Examples

```
>>> G = nx.Graph()
>>> G.add_edge(1,2)
>>> G.add_node(3)
>>> nx.isolates(G)
[3]
```

To remove all isolates in the graph use >>> G.remove_nodes_from(nx.isolates(G)) >>> G.nodes() [1, 2]

For digraphs isolates have zero in-degree and zero out_degre >>> G = nx.DiGraph([(0,1),(1,2)]) >>> G.add_node(3) >>> nx.isolates(G) [3]

4.17 Isomorphism

is_isomorphic(G1, G2[, weighted, rtol, atol])	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.

4.17.1 networkx.algorithms.isomorphism.is_isomorphic

networkx.algorithms.isomorphism.**is_isomorphic**(*G1*, *G2*, *weighted=False*, *rtol=9.9999999999999995e-07*, *atol=1.0000000000000001e-09*)

Returns True if the graphs G1 and G2 are isomorphic and False otherwise.

Parameters **G1, G2: NetworkX graph instances** :

The two graphs G1 and G2 must be the same type.

weighted: bool, optional :

Optionally check isomorphism for weighted graphs. G1 and G2 must be valid weighted graphs.

rtol: float, optional :

The relative error tolerance when checking weighted edges

atol: float, optional :

The absolute error tolerance when checking weighted edges

See Also:

isomorphvf2

Notes

Uses the vf2 algorithm. Works for Graph, DiGraph, MultiGraph, and MultiDiGraph

4.17.2 networkx.algorithms.isomorphism.could_be_isomorphic

networkx.algorithms.isomorphism.**could_be_isomorphic**(*G1*, *G2*)

Returns False if graphs are definitely not isomorphic. True does NOT guarantee isomorphism.

Parameters **G1, G2** : NetworkX graph instances

The two graphs G1 and G2 must be the same type.

Notes

Checks for matching degree, triangle, and number of cliques sequences.

4.17.3 networkx.algorithms.isomorphism.fast_could_be_isomorphic

networkx.algorithms.isomorphism.**fast_could_be_isomorphic**(*G1*, *G2*)

Returns False if graphs are definitely not isomorphic. True does NOT guarantee isomorphism.

Parameters **G1, G2** : NetworkX graph instances

The two graphs G1 and G2 must be the same type.

Notes

Checks for matching degree and triangle sequences.

4.17.4 networkx.algorithms.isomorphism.faster_could_be_isomorphic

networkx.algorithms.isomorphism.**faster_could_be_isomorphic**(*G1*, *G2*)

Returns False if graphs are definitely not isomorphic. True does NOT guarantee isomorphism.

Parameters **G1, G2** : NetworkX graph instances

The two graphs G1 and G2 must be the same type.

Notes

Checks for matching degree sequences.

4.17.5 Advanced Interface to VF2 Algorithm

VF2 Algorithm

Graph Matcher

GraphMatcher.__init__(G1, G2)	Initialize GraphMatcher.
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 adding (G1_node, G2_node) is symantically feasible.
GraphMatcher.syntactic_feasibility(G1_node, ...)	Returns True if adding (G1_node, G2_node) is syntactically feasible.

networkx.GraphMatcher.__init__
GraphMatcher.**__init__**(*G1*, *G2*)
Initialize GraphMatcher.

Parameters **G1,G2: NetworkX Graph or MultiGraph instances.** :

The two graphs to check for isomorphism.

Examples

To create a GraphMatcher which checks for syntactic feasibility:

```
>>> G1 = nx.path_graph(4)
>>> G2 = nx.path_graph(4)
>>> GM = nx.GraphMatcher(G1,G2)
```

networkx.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.

networkx.GraphMatcher.is_isomorphic
GraphMatcher.**is_isomorphic**()
Returns True if G1 and G2 are isomorphic graphs.

networkx.GraphMatcher.subgraph_is_isomorphic
GraphMatcher.**subgraph_is_isomorphic**()
Returns True if a subgraph of G1 is isomorphic to G2.

networkx.GraphMatcher.isomorphisms_iter
GraphMatcher.**isomorphisms_iter**()
Generator over isomorphisms between G1 and G2.

networkx.GraphMatcher.subgraph_isomorphisms_iter
GraphMatcher.**subgraph_isomorphisms_iter**()
Generator over isomorphisms between a subgraph of G1 and G2.

networkx.GraphMatcher.candidate_pairs_iter
GraphMatcher.**candidate_pairs_iter**()
Iterator over candidate pairs of nodes in G1 and G2.

networkx.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.

networkx.GraphMatcher.semantic_feasibility

GraphMatcher.**semantic_feasibility**(*G1_node*, *G2_node*)

Returns True if adding (G1_node, G2_node) is symantically feasible.

The semantic feasibility function should return True if it is acceptable to add the candidate pair (G1_node, G2_node) to the current partial isomorphism mapping. The logic should focus on semantic information contained in the edge data or a formalized node class.

By acceptable, we mean that the subsequent mapping can still become a complete isomorphism mapping. Thus, if adding the candidate pair definitely makes it so that the subsequent mapping cannot become a complete isomorphism mapping, then this function must return False.

The default semantic feasibility function always returns True. The effect is that semantics are not considered in the matching of G1 and G2.

The semantic checks might differ based on the what type of test is being performed. A keyword description of the test is stored in self.test. Here is a quick description of the currently implemented tests:

test='graph' Indicates that the graph matcher is looking for a graph-graph isomorphism.

test='subgraph' Indicates that the graph matcher is looking for a subgraph-graph isomorphism such that a subgraph of G1 is isomorphic to G2.

Any subclass which redefines semantic_feasibility() must maintain the above form to keep the match() method functional. Implementations should consider multigraphs.

networkx.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 mapping is allowable. The addition is allowable if the inclusion of the candidate pair does not make it impossible for an isomorphism to be found.

DiGraph Matcher

DiGraphMatcher.__init__(G1, G2)	Initialize DiGraphMatcher.
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 adding (G1_node, G2_node) is symantically feasible.
DiGraphMatcher.syntactic_feasibility(...)	Returns True if adding (G1_node, G2_node) is syntactically feasible.

networkx.DiGraphMatcher.__init__

DiGraphMatcher.**__init__**(*G1*, *G2*)

Initialize DiGraphMatcher.

G1 and G2 should be nx.Graph or nx.MultiGraph instances.

Examples

To create a GraphMatcher which checks for syntactic feasibility:

```
>>> G1 = nx.DiGraph(nx.path_graph(4, create_using=nx.DiGraph()))
>>> G2 = nx.DiGraph(nx.path_graph(4, create_using=nx.DiGraph()))
>>> DiGM = nx.DiGraphMatcher(G1,G2)
```

networkx.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.

networkx.DiGraphMatcher.is_isomorphic

DiGraphMatcher.**is_isomorphic**()

Returns True if G1 and G2 are isomorphic graphs.

networkx.DiGraphMatcher.subgraph_is_isomorphic

DiGraphMatcher.**subgraph_is_isomorphic**()

Returns True if a subgraph of G1 is isomorphic to G2.

networkx.DiGraphMatcher.isomorphisms_iter

DiGraphMatcher.**isomorphisms_iter**()

Generator over isomorphisms between G1 and G2.

networkx.DiGraphMatcher.subgraph_isomorphisms_iter

DiGraphMatcher.**subgraph_isomorphisms_iter**()

Generator over isomorphisms between a subgraph of G1 and G2.

networkx.DiGraphMatcher.candidate_pairs_iter

DiGraphMatcher.**candidate_pairs_iter**()

Iterator over candidate pairs of nodes in G1 and G2.

networkx.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.

networkx.DiGraphMatcher.semantic_feasibility

DiGraphMatcher.**semantic_feasibility**(*G1_node*, *G2_node*)

Returns True if adding (G1_node, G2_node) is symantically feasible.

The semantic feasibility function should return True if it is acceptable to add the candidate pair (G1_node, G2_node) to the current partial isomorphism mapping. The logic should focus on semantic information contained in the edge data or a formalized node class.

By acceptable, we mean that the subsequent mapping can still become a complete isomorphism mapping. Thus, if adding the candidate pair definitely makes it so that the subsequent mapping cannot become a complete isomorphism mapping, then this function must return False.

The default semantic feasibility function always returns True. The effect is that semantics are not considered in the matching of G1 and G2.

The semantic checks might differ based on the what type of test is being performed. A keyword description of the test is stored in self.test. Here is a quick description of the currently implemented tests:

test='graph' Indicates that the graph matcher is looking for a graph-graph isomorphism.

test='subgraph' Indicates that the graph matcher is looking for a subgraph-graph isomorphism such that a subgraph of G1 is isomorphic to G2.

Any subclass which redefines semantic_feasibility() must maintain the above form to keep the match() method functional. Implementations should consider multigraphs.

networkx.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 mapping is allowable. The addition is allowable if the inclusion of the candidate pair does not make it impossible for an isomorphism to be found.

Weighted Graph Matcher

WeightedGraphMatcher.__init__(G1, G2[, ...])	Initialize WeightedGraphMatcher.
WeightedGraphMatcher.initialize()	Reinitializes the state of the algorithm.
WeightedGraphMatcher.is_isomorphic()	Returns True if G1 and G2 are isomorphic graphs.
WeightedGraphMatcher.subgraph_is_isomorphic()	Returns True if a subgraph of G1 is isomorphic to G2.
WeightedGraphMatcher.isomorphisms_iter()	Generator over isomorphisms between G1 and G2.
WeightedGraphMatcher.subgraph_isomorphisms_iter()	Generator over isomorphisms between a subgraph of G1 and G2.
WeightedGraphMatcher.candidate_pairs_iter()	Iterator over candidate pairs of nodes in G1 and G2.
WeightedGraphMatcher.match()	Extends the isomorphism mapping.
WeightedGraphMatcher.semantic_feasibility()	Returns True if mapping G1_node to G2_node is semantically feasible.
WeightedGraphMatcher.syntactic_feasibility()	Returns True if adding (G1_node, G2_node) is syntactically feasible.

networkx.WeightedGraphMatcher.__init__

WeightedGraphMatcher.**__init__**(*G1*, *G2*, *rtol=9.9999999999999995e-07*, *atol=1.0000000000000001e-09*)

Initialize WeightedGraphMatcher.

Parameters **G1, G2** : nx.Graph instances

G1 and G2 must be weighted graphs.

rtol : float, optional

The relative tolerance used to compare weights.

atol : float, optional

The absolute tolerance used to compare weights.

networkx.WeightedGraphMatcher.initialize
WeightedGraphMatcher.**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.

networkx.WeightedGraphMatcher.is_isomorphic
WeightedGraphMatcher.**is_isomorphic**()
Returns True if G1 and G2 are isomorphic graphs.

networkx.WeightedGraphMatcher.subgraph_is_isomorphic
WeightedGraphMatcher.**subgraph_is_isomorphic**()
Returns True if a subgraph of G1 is isomorphic to G2.

networkx.WeightedGraphMatcher.isomorphisms_iter
WeightedGraphMatcher.**isomorphisms_iter**()
Generator over isomorphisms between G1 and G2.

networkx.WeightedGraphMatcher.subgraph_isomorphisms_iter
WeightedGraphMatcher.**subgraph_isomorphisms_iter**()
Generator over isomorphisms between a subgraph of G1 and G2.

networkx.WeightedGraphMatcher.candidate_pairs_iter
WeightedGraphMatcher.**candidate_pairs_iter**()
Iterator over candidate pairs of nodes in G1 and G2.

networkx.WeightedGraphMatcher.match
WeightedGraphMatcher.**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.

networkx.WeightedGraphMatcher.semantic_feasibility
WeightedGraphMatcher.**semantic_feasibility**(*G1_node*, *G2_node*)
Returns True if mapping G1_node to G2_node is semantically feasible.

networkx.WeightedGraphMatcher.syntactic_feasibility
WeightedGraphMatcher.**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 mapping is allowable. The addition is allowable if the inclusion of the candidate pair does not make it impossible for an isomorphism to be found.

Weighted DiGraph Matcher

WeightedDiGraphMatcher.__init__(G1, G2[, ...])	Initialize WeightedGraphMatcher.
WeightedDiGraphMatcher.initialize()	Reinitializes the state of the algorithm.
WeightedDiGraphMatcher.is_isomorphic()	Returns True if G1 and G2 are isomorphic graphs.
WeightedDiGraphMatcher.subgraph_is_isomorphic()	Returns True if a subgraph of G1 is isomorphic to G2.
WeightedDiGraphMatcher.isomorphisms_iter()	Generator over isomorphisms between G1 and G2.
WeightedDiGraphMatcher.subgraph_isomorphisms_iter()	Generator over isomorphisms between a subgraph of G1 and G2.
WeightedDiGraphMatcher.candidate_pairs_iter()	Iterator over candidate pairs of nodes in G1 and G2.
WeightedDiGraphMatcher.match()	Extends the isomorphism mapping.
WeightedDiGraphMatcher.semantic_feasibility(n, ...)	Returns True if mapping G1_node to G2_node is semantically feasible.
WeightedDiGraphMatcher.syntactic_feasibility(n, ...)	Returns True if adding (G1_node, G2_node) is syntactically feasible.

networkx.WeightedDiGraphMatcher.__init__

WeightedDiGraphMatcher.**__init__**(*G1*, *G2*, *rtol=9.9999999999999995e-07*, *atol=1.0000000000000001e-09*)

Initialize WeightedGraphMatcher.

Parameters **G1, G2** : nx.DiGraph instances

G1 and G2 must be weighted graphs.

rtol : float, optional

The relative tolerance used to compare weights.

atol : float, optional

The absolute tolerance used to compare weights.

networkx.WeightedDiGraphMatcher.initialize

WeightedDiGraphMatcher.**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.

networkx.WeightedDiGraphMatcher.is_isomorphic

WeightedDiGraphMatcher.**is_isomorphic**()

Returns True if G1 and G2 are isomorphic graphs.

networkx.WeightedDiGraphMatcher.subgraph_is_isomorphic

WeightedDiGraphMatcher.**subgraph_is_isomorphic**()

Returns True if a subgraph of G1 is isomorphic to G2.

networkx.WeightedDiGraphMatcher.isomorphisms_iter

WeightedDiGraphMatcher.**isomorphisms_iter**()

Generator over isomorphisms between G1 and G2.

networkx.WeightedDiGraphMatcher.subgraph_isomorphisms_iter

WeightedDiGraphMatcher.**subgraph_isomorphisms_iter**()

Generator over isomorphisms between a subgraph of G1 and G2.

networkx.WeightedDiGraphMatcher.candidate_pairs_iter

WeightedDiGraphMatcher.**candidate_pairs_iter**()

Iterator over candidate pairs of nodes in G1 and G2.

networkx.WeightedDiGraphMatcher.match

WeightedDiGraphMatcher.**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.

networkx.WeightedDiGraphMatcher.semantic_feasibility

WeightedDiGraphMatcher.**semantic_feasibility**(*G1_node*, *G2_node*)

Returns True if mapping G1_node to G2_node is semantically feasible.

networkx.WeightedDiGraphMatcher.syntactic_feasibility

WeightedDiGraphMatcher.**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 mapping is allowable. The addition is allowable if the inclusion of the candidate pair does not make it impossible for an isomorphism to be found.

Weighted MultiGraph Matcher

WeightedMultiGraphMatcher.__init__(G1, G2[, ...])	Initialize WeightedGraphMatcher.
WeightedMultiGraphMatcher.initialize()	Reinitializes the state of the algorithm.
WeightedMultiGraphMatcher.is_isomorphic()	Returns True if G1 and G2 are isomorphic graphs.
WeightedMultiGraphMatcher.subgraph_is_isomorphic()	Returns True if a subgraph of G1 is isomorphic to G2.
WeightedMultiGraphMatcher.isomorphisms_iter()	Generator over isomorphisms between G1 and G2.
WeightedMultiGraphMatcher.subgraph_isomorphisms_iter()	Generator over isomorphisms between a subgraph of G1 and G2.
WeightedMultiGraphMatcher.candidate_pairs_iter()	Iterator over candidate pairs of nodes in G1 and G2.
WeightedMultiGraphMatcher.match()	Extends the isomorphism mapping.
WeightedMultiGraphMatcher.semantic_feasibility()	Returns True if mapping G1_node to G2_node is semantically feasible.
WeightedMultiGraphMatcher.syntactic_feasibility()	Returns True if adding (G1_node, G2_node) is syntactically feasible.

networkx.WeightedMultiGraphMatcher.__init__

WeightedMultiGraphMatcher.**__init__**(*G1*, *G2*, *rtol=9.9999999999999995e-07*, *atol=1.0000000000000001e-09*)

Initialize WeightedGraphMatcher.

Parameters **G1, G2** : nx.MultiGraph instances

G1 and G2 must be weighted graphs.

rtol : float, optional

The relative tolerance used to compare weights.

atol : float, optional

The absolute tolerance used to compare weights.

networkx.WeightedMultiGraphMatcher.initialize
`WeightedMultiGraphMatcher.`**`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.

networkx.WeightedMultiGraphMatcher.is_isomorphic
`WeightedMultiGraphMatcher.`**`is_isomorphic`**`()`
Returns True if G1 and G2 are isomorphic graphs.

networkx.WeightedMultiGraphMatcher.subgraph_is_isomorphic
`WeightedMultiGraphMatcher.`**`subgraph_is_isomorphic`**`()`
Returns True if a subgraph of G1 is isomorphic to G2.

networkx.WeightedMultiGraphMatcher.isomorphisms_iter
`WeightedMultiGraphMatcher.`**`isomorphisms_iter`**`()`
Generator over isomorphisms between G1 and G2.

networkx.WeightedMultiGraphMatcher.subgraph_isomorphisms_iter
`WeightedMultiGraphMatcher.`**`subgraph_isomorphisms_iter`**`()`
Generator over isomorphisms between a subgraph of G1 and G2.

networkx.WeightedMultiGraphMatcher.candidate_pairs_iter
`WeightedMultiGraphMatcher.`**`candidate_pairs_iter`**`()`
Iterator over candidate pairs of nodes in G1 and G2.

networkx.WeightedMultiGraphMatcher.match
`WeightedMultiGraphMatcher.`**`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.

networkx.WeightedMultiGraphMatcher.semantic_feasibility
`WeightedMultiGraphMatcher.`**`semantic_feasibility`**`(`*`G1_node`*`, `*`G2_node`*`)`
Returns True if mapping G1_node to G2_node is semantically feasible.

networkx.WeightedMultiGraphMatcher.syntactic_feasibility

WeightedMultiGraphMatcher.**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 mapping is allowable. The addition is allowable if the inclusion of the candidate pair does not make it impossible for an isomorphism to be found.

Weighted MultiDiGraph Matcher

WeightedMultiDiGraphMatcher.__init__(G1, G2)	Initialize WeightedGraphMatcher.
WeightedMultiDiGraphMatcher.initialize()	Reinitializes the state of the algorithm.
WeightedMultiDiGraphMatcher.is_isomorphic()	Returns True if G1 and G2 are isomorphic graphs.
WeightedMultiDiGraphMatcher.subgraph_is_isomorphic()	Returns True if a subgraph of G1 is isomorphic to G2.
WeightedMultiDiGraphMatcher.isomorphisms_iter()	Generator over isomorphisms between G1 and G2.
WeightedMultiDiGraphMatcher.subgraph_isomorphisms_iter()	Generator over isomorphisms between a subgraph of G1 and G2.
WeightedMultiDiGraphMatcher.candidate_pairs_iter()	Iterator over candidate pairs of nodes in G1 and G2.
WeightedMultiDiGraphMatcher.match()	Extends the isomorphism mapping.
WeightedMultiDiGraphMatcher.semantic_feasibility()	Returns True if mapping G1_node to G2_node is semantically feasible.
WeightedMultiDiGraphMatcher.syntactic_feasibility()	Returns True if adding (G1_node, G2_node) is syntactically feasible.

networkx.WeightedMultiDiGraphMatcher.__init__

WeightedMultiDiGraphMatcher.**__init__**(*G1*, *G2*, *rtol=9.9999999999999995e-07*, *atol=1.0000000000000001e-09*)

Initialize WeightedGraphMatcher.

Parameters **G1, G2** : nx.MultiDiGraph instances

G1 and G2 must be weighted graphs.

rtol : float, optional

The relative tolerance used to compare weights.

atol : float, optional

The absolute tolerance used to compare weights.

networkx.WeightedMultiDiGraphMatcher.initialize

WeightedMultiDiGraphMatcher.**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.

networkx.WeightedMultiDiGraphMatcher.is_isomorphic

WeightedMultiDiGraphMatcher.**is_isomorphic**()

Returns True if G1 and G2 are isomorphic graphs.

networkx.WeightedMultiDiGraphMatcher.subgraph_is_isomorphic
WeightedMultiDiGraphMatcher.**subgraph_is_isomorphic**()
Returns True if a subgraph of G1 is isomorphic to G2.

networkx.WeightedMultiDiGraphMatcher.isomorphisms_iter
WeightedMultiDiGraphMatcher.**isomorphisms_iter**()
Generator over isomorphisms between G1 and G2.

networkx.WeightedMultiDiGraphMatcher.subgraph_isomorphisms_iter
WeightedMultiDiGraphMatcher.**subgraph_isomorphisms_iter**()
Generator over isomorphisms between a subgraph of G1 and G2.

networkx.WeightedMultiDiGraphMatcher.candidate_pairs_iter
WeightedMultiDiGraphMatcher.**candidate_pairs_iter**()
Iterator over candidate pairs of nodes in G1 and G2.

networkx.WeightedMultiDiGraphMatcher.match
WeightedMultiDiGraphMatcher.**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.

networkx.WeightedMultiDiGraphMatcher.semantic_feasibility
WeightedMultiDiGraphMatcher.**semantic_feasibility**(*G1_node*, *G2_node*)
Returns True if mapping G1_node to G2_node is semantically feasible.

networkx.WeightedMultiDiGraphMatcher.syntactic_feasibility
WeightedMultiDiGraphMatcher.**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 mapping is allowable. The addition is allowable if the inclusion of the candidate pair does not make it impossible for an isomorphism to be found.

4.18 Link Analysis

4.18.1 PageRank

PageRank analysis of graph structure.

pagerank(G[, alpha, personalization, ...])	Return the PageRank of the nodes in the graph.
pagerank_numpy(G[, alpha, personalization])	Return the PageRank of the nodes in the graph.
pagerank_scipy(G[, alpha, personalization, ...])	Return the PageRank of the nodes in the graph.
google_matrix(G[, alpha, personalization, ...])	Return the Google matrix of the graph.

networkx.algorithms.link_analysis.pagerank_alg.pagerank

networkx.algorithms.link_analysis.pagerank_alg.**pagerank**(*G*, *alpha=0.84999999999999998*, *personalization=None*, *max_iter=100*, *tol=1e-08*, *nstart=None*)

Return 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

alpha : float, optional

Damping parameter for PageRank, default=0.85

personalization: dict, optional :

The "personalization vector" consisting of a dictionary with a key for every graph node and nonzero personalization value for each node.

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.

Returns **pagerank** : dictionary

Dictionary of nodes with PageRank as value

See Also:

pagerank_numpy, pagerank_scipy, google_matrix

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 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 oriented edge in the directed graph to two edges.

References

[R129], [R130]

Examples

```
>>> G=nx.DiGraph(nx.path_graph(4))
>>> pr=nx.pagerank(G,alpha=0.9)
```

networkx.algorithms.link_analysis.pagerank_alg.pagerank_numpy

networkx.algorithms.link_analysis.pagerank_alg.**pagerank_numpy**(*G*, *alpha=0.84999999999999998*, *personalization=None*)

Return 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

alpha : float, optional

Damping parameter for PageRank, default=0.85

personalization: dict, optional :

The "personalization vector" consisting of a dictionary with a key for every graph node and nonzero personalization value for each node.

Returns **pagerank** : dictionary

Dictionary of nodes with PageRank as value

See Also:

pagerank, pagerank_scipy, google_matrix

Notes

The eigenvector calculation uses NumPy's interface to the LAPACK eigenvalue solvers. This will be the fastest and most accurate for small graphs.

This implementation works with Multi(Di)Graphs.

References

[R131], [R132]

Examples

```
>>> G=nx.DiGraph(nx.path_graph(4))
>>> pr=nx.pagerank_numpy(G,alpha=0.9)
```

networkx.algorithms.link_analysis.pagerank_alg.pagerank_scipy

networkx.algorithms.link_analysis.pagerank_alg.**pagerank_scipy**(*G*, *alpha=0.84999999999999998*, *personalization=None*, *max_iter=100*, *tol=9.9999999999999995e-07*)

Return 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

alpha : float, optional

Damping parameter for PageRank, default=0.85

personalization: dict, optional :

The "personalization vector" consisting of a dictionary with a key for every graph node and nonzero personalization value for each node.

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.

Returns **pagerank** : dictionary

Dictionary of nodes with PageRank as value

See Also:

pagerank, pagerank_numpy, google_matrix

Notes

The eigenvector calculation uses power iteration with a SciPy sparse matrix representation.

References

[R133], [R134]

Examples

```
>>> G=nx.DiGraph(nx.path_graph(4))
>>> pr=nx.pagerank_scipy(G,alpha=0.9)
```

networkx.algorithms.link_analysis.pagerank_alg.google_matrix

networkx.algorithms.link_analysis.pagerank_alg.**google_matrix**(*G*, *alpha=0.84999999999999998*, *personalization=None*, *nodelist=None*)

Return the Google matrix of the graph.

Parameters **G** : graph

A NetworkX graph

alpha : float

The damping factor

personalization: dict, optional :

The "personalization vector" consisting of a dictionary with a key for every graph node and nonzero personalization value for each node.

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 **A** : NumPy matrix

Google matrix of the graph

See Also:

pagerank, pagerank_numpy, pagerank_scipy

4.18.2 Hits

Hubs and authorities analysis of graph structure.

hits(G[, max_iter, tol, nstart])	Return HITS hubs and authorities values for nodes.
hits_numpy(G)	Return HITS hubs and authorities values for nodes.
hits_scipy(G[, max_iter, tol])	Return HITS hubs and authorities values for nodes.
hub_matrix(G[, nodelist])	Return the HITS hub matrix.
authority_matrix(G[, nodelist])	Return the HITS authority matrix.

networkx.algorithms.link_analysis.hits_alg.hits

networkx.algorithms.link_analysis.hits_alg.**hits**(*G*, *max_iter=100*, *tol=1e-08*, *nstart=None*)

Return 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 : interger, 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.

Returns **(hubs,authorities)** : two-tuple of dictionaries

Two dictionaries keyed by node containing the hub and authority values.

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

[R123], [R124]

Examples

```
>>> G=nx.path_graph(4)
>>> h,a=nx.hits(G)
```

networkx.algorithms.link_analysis.hits_alg.hits_numpy

networkx.algorithms.link_analysis.hits_alg.**hits_numpy**(*G*)

Return 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

Returns **(hubs,authorities)** : two-tuple of dictionaries

Two dictionaries keyed by node containing the hub and authority values.

Notes

The eigenvector calculation uses NumPy's interface to LAPACK.

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

[R125], [R126]

Examples

```
>>> G=nx.path_graph(4)
>>> h,a=nx.hits(G)
```

networkx.algorithms.link_analysis.hits_alg.hits_scipy

networkx.algorithms.link_analysis.hits_alg.**hits_scipy**(*G*, *max_iter=100*, *tol=9.9999999999999995e-07*)

Return 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 : interger, 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.

Returns **(hubs,authorities)** : two-tuple of dictionaries

Two dictionaries keyed by node containing the hub and authority values.

Notes

This implementation uses SciPy sparse matrices.

The eigenvector calculation is done by the power iteration method and has no guarantee of convergence. The iteration will stop after max_iter iterations or an error tolerance of number_of_nodes(G)*tol has been reached.

The HITS algorithm was designed for directed graphs but this algorithm does not check if the input graph is directed and will execute on undirected graphs.

References

[R127], [R128]

Examples

```
>>> G=nx.path_graph(4)
>>> h,a=nx.hits(G)
```

networkx.algorithms.link_analysis.hits_alg.hub_matrix

networkx.algorithms.link_analysis.hits_alg.**hub_matrix**(*G*, *nodelist=None*)
Return the HITS hub matrix.

networkx.algorithms.link_analysis.hits_alg.authority_matrix

networkx.algorithms.link_analysis.hits_alg.**authority_matrix**(*G*, *nodelist=None*)
Return the HITS authority matrix.

4.19 Matching

The algorithm is taken from "Efficient Algorithms for Finding Maximum Matching in Graphs" by Zvi Galil, ACM Computing Surveys, 1986. It 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.

max_weight_matching(G[, maxcardinality])	Compute a maximum-weighted matching of G.

4.19.1 networkx.algorithms.matching.max_weight_matching

networkx.algorithms.matching.**max_weight_matching**(*G*, *maxcardinality=False*)
Compute a maximum-weighted matching of G.

A matching is a subset of edges in which no node occurs more than once. The cardinality of a matching is the number of matched edges. The weight of a matching is the sum of the weights of its edges.

Parameters **G** : NetworkX graph

Undirected graph

maxcardinality: bool, optional :

If maxcardinality is True, compute the maximum-cardinality matching with maximum weight among all maximum-cardinality matchings.

Returns **mate** : dictionary

The matching is returned as a dictionary, mate, such that mate[v] == w if node v is matched to node w. Unmatched nodes do not occur as a key in mate.

Notes

If G has edges with 'weight' attribute 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.

References

[R135]

4.20 Mixing Patterns

Mixing matrices and assortativity coefficients.

4.20.1 Assortativity

degree_assortativity(G[, nodes])	Compute degree assortativity of graph.
attribute_assortativity(G, attribute[, nodes])	Compute assortativity for node attributes.
numeric_assortativity(G, attribute[, nodes])	Compute assortativity for numerical node attributes.
degree_pearsonr(G[, nodes])	Compute degree assortativity of graph.

networkx.algorithms.mixing.degree_assortativity

networkx.algorithms.mixing.**degree_assortativity**(*G*, *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

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, numeric_assortativity, neighbor_connectivity, degree_mixing_dict, degree_mixing_matrix

Notes

This computes Eq. (21) in Ref. [R137] , 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 out-degree and in-degree.

References

[R137]

Examples

```
>>> G=nx.path_graph(4)
>>> r=nx.degree_assortativity(G)
>>> print("%3.1f"%r)
-0.5
```

networkx.algorithms.mixing.attribute_assortativity

networkx.algorithms.mixing.**attribute_assortativity**(*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 **a: float** :

Assortativity of given attribute

Notes

This computes Eq. (2) in Ref. [R136] , (trace(e)-sum(e))/(1-sum(e)), where e is the joint probability distribution (mixing matrix) of the specified attribute.

References

[R136]

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(G,'color'))
1.0
```

networkx.algorithms.mixing.numeric_assortativity

networkx.algorithms.mixing.**numeric_assortativity**(*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 **a: float** :

Assortativity of given attribute

Notes

This computes Eq. (21) in Ref. [R139] , where e is the joint probability distribution (mixing matrix) of the specified attribute.

References

[R139]

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(G,'size'))
1.0
```

networkx.algorithms.mixing.degree_pearsonr

networkx.algorithms.mixing.**degree_pearsonr**(*G*, *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

nodes: list or iterable (optional) :

Compute pearson correlation of degrees only for nodes in container. The default is all nodes.

Returns **r** : float

Assortativity of graph by degree.

Notes

This calls scipy.stats.pearsonr().

References

[R138]

Examples

```
>>> G=nx.path_graph(4)
>>> r=nx.degree_pearsonr(G)
>>> r
-0.5
```

4.20.2 Mixing

attribute_mixing_matrix(G, attribute[, ...])	Return mixing matrix for attribute.
degree_mixing_matrix(G[, nodes, normalized])	Return mixing matrix for attribute.
degree_mixing_dict(G[, nodes, normalized])	Return dictionary representation of mixing matrix for degree.
attribute_mixing_dict(G, attribute[, nodes, ...])	Return dictionary representation of mixing matrix for attribute.

networkx.algorithms.mixing.attribute_mixing_matrix

networkx.algorithms.mixing.**attribute_mixing_matrix**(*G*, *attribute*, *nodes=None*, *mapping=None*, *normalized=True*)

Return 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=False)

Return counts if False or probabilities if True.

Returns **m: numpy array** :

Counts or joint probability of occurrence of attribute pairs.

networkx.algorithms.mixing.degree_mixing_matrix

networkx.algorithms.mixing.**degree_mixing_matrix**(*G*, *nodes=None*, *normalized=True*)

Return mixing matrix for attribute.

Parameters **G** : graph

NetworkX graph object.

nodes: list or iterable (optional) :

Build the matrix using only nodes in container. The default is all nodes.

normalized : bool (default=False)

Return counts if False or probabilities if True.

Returns **m: numpy array** :

Counts, or joint probability, of occurrence of node degree.

networkx.algorithms.mixing.degree_mixing_dict

networkx.algorithms.mixing.**degree_mixing_dict**(*G*, *nodes=None*, *normalized=False*)

Return dictionary representation of mixing matrix for degree.

Parameters **G** : graph

NetworkX graph object.

normalized : bool (default=False)

Return counts if False or probabilities if True.

Returns **d: dictionary** :

Counts or joint probability of occurrence of degree pairs.

networkx.algorithms.mixing.attribute_mixing_dict

networkx.algorithms.mixing.**attribute_mixing_dict**(*G*, *attribute*, *nodes=None*, *normalized=False*)

Return 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
>>> print(d['blue']['red']) # d symmetric for undirected graphs
1
```

4.21 Maximal independent set

Algorithm to find a maximal (not maximum) independent set.

maximal_independent_set(G[, nodes])	Return a random maximal independent set guaranteed to contain a given set of nodes.

4.21.1 networkx.algorithms.mis.maximal_independent_set

networkx.algorithms.mis.**maximal_independent_set**(*G*, *nodes=None*)

Return 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.

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.

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]
```

4.22 Minimum Spanning Tree

Computes minimum spanning tree of a weighted graph.

minimum_spanning_tree(G[, weight])	Return a minimum spanning tree or forest of an undirected weighted graph.
minimum_spanning_edges(G[, weight, data])	Generate edges in a minimum spanning forest of an undirected weighted graph.

4.22.1 networkx.algorithms.mst.minimum_spanning_tree

networkx.algorithms.mst.**minimum_spanning_tree**(*G*, *weight='weight'*)

Return a minimum spanning tree or 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.

If the graph is not connected a spanning forest is constructed. A spanning forest is a union of the spanning trees for each connected component of the graph.

Parameters **G** : NetworkX Graph

weight : string

Edge data key to use for weight (default 'weight').

Returns **G** : NetworkX Graph

A minimum spanning tree or forest.

Notes

Uses Kruskal's algorithm.

If the graph edges do not have a weight attribute a default weight of 1 will be used.

Examples

```
>>> G=nx.cycle_graph(4)
>>> G.add_edge(0,3,weight=2) # assign weight 2 to edge 0-3
>>> T=nx.minimum_spanning_tree(G)
>>> print(sorted(T.edges(data=True)))
[(0, 1, {}), (1, 2, {}), (2, 3, {})]
```

4.22.2 networkx.algorithms.mst.minimum_spanning_edges

networkx.algorithms.mst.**minimum_spanning_edges**(*G*, *weight='weight'*, *data=True*)

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** : NetworkX Graph

weight : string

Edge data key to use for weight (default 'weight').

data : bool, optional

If True yield the edge data along with the edge.

Returns **edges** : iterator

A generator that produces edges in the minimum spanning tree. The edges are three-tuples (u,v,w) where w is the weight.

Notes

Uses Kruskal's algorithm.

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

```
>>> G=nx.cycle_graph(4)
>>> G.add_edge(0,3,weight=2) # assign weight 2 to edge 0-3
>>> mst=nx.minimum_spanning_edges(G,data=False) # a generator of MST edges
>>> edgelist=list(mst) # make a list of the edges
>>> print(sorted(edgelist))
[(0, 1), (1, 2), (2, 3)]
```

4.23 Operators

Operations on graphs including union, intersection, difference, complement, subgraph.

cartesian_product(G, H[, create_using])	Return the Cartesian product of G and H.
compose(G, H[, create_using, name])	Return a new graph of G composed with H.
complement(G[, create_using, name])	Return graph complement of G.
union(G, H[, create_using, rename, name])	Return the union of graphs G and H.
disjoint_union(G, H)	Return the disjoint union of graphs G and H, forcing distinct integer node labels.
intersection(G, H[, create_using])	Return a new graph that contains only the edges that exist in both G and H.
difference(G, H[, create_using])	Return a new graph that contains the edges that exist in G
symmetric_difference(G, H[, create_using])	Return new graph with edges that exist in either G or H but not both.

4.23.1 networkx.algorithms.operators.cartesian_product

networkx.algorithms.operators.**cartesian_product**(*G*, *H*, *create_using=None*)

Return the Cartesian product of G and H.

Parameters **G,H** : graph

A NetworkX graph

create_using : NetworkX graph

Use specified graph for result. Otherwise a new graph is created with the same type as G.

Notes

Only tested with Graph class. Graph, node, and edge attributes are not copied to the new graph.

4.23.2 networkx.algorithms.operators.compose

networkx.algorithms.operators.**compose**(*G*, *H*, *create_using=None*, *name=None*)
Return a new graph of G composed with H.

Composition is the simple union of the node sets and edge sets. The node sets of G and H need not be disjoint.

Parameters **G,H** : graph

A NetworkX graph

create_using : NetworkX graph

Use specified graph for result. Otherwise a new graph is created with the same type as G

name : string

Specify name for new graph

Notes

A new graph is returned, of the same class as G. It is recommended that G and H be either both directed or both undirected. Attributes from G take precedent over attributes from H.

4.23.3 networkx.algorithms.operators.complement

networkx.algorithms.operators.**complement**(*G*, *create_using=None*, *name=None*)
Return graph complement of G.

Parameters **G** : graph

A NetworkX graph

create_using : NetworkX graph

Use specified graph for result. Otherwise a new graph is created.

name : string

Specify name for 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.

4.23.4 networkx.algorithms.operators.union

networkx.algorithms.operators.**union**(*G*, *H*, *create_using=None*, *rename=False*, *name=None*)
Return the union of graphs G and H.

Graphs G and H must be disjoint, otherwise an exception is raised.

Parameters **G,H** : graph

A NetworkX graph

create_using : NetworkX graph

Use specified graph for result. Otherwise a new graph is created with the same type as G.

rename : bool (default=False)

Node names of G and H can be changed be 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".

name : string

Specify the name for the union graph

See Also:

`disjoint_union`

Notes

To force a disjoint union with node relabeling, use disjoint_union(G,H) or convert_node_labels_to integers().

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 G is used.

4.23.5 networkx.algorithms.operators.disjoint_union

networkx.algorithms.operators.**disjoint_union**(*G*, *H*)

Return the disjoint union of graphs G and H, forcing distinct integer node labels.

Parameters **G,H** : graph

A NetworkX graph

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.

4.23.6 networkx.algorithms.operators.intersection

networkx.algorithms.operators.**intersection**(*G*, *H*, *create_using=None*)

Return a new graph that contains only the edges that exist in both G and 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.

create_using : NetworkX graph

Use specified graph for result. Otherwise a new graph is created 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 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)
```

4.23.7 networkx.algorithms.operators.difference

networkx.algorithms.operators.**difference**(*G*, *H*, *create_using=None*)

Return 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.

create_using : NetworkX graph

Use specified graph for result. Otherwise a new graph is created 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 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)
```

4.23.8 networkx.algorithms.operators.symmetric_difference

networkx.algorithms.operators.**symmetric_difference**(*G*, *H*, *create_using=None*)

Return 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.

create_using : NetworkX graph

Use specified graph for result. Otherwise a new graph is created with the same type as G.

Notes

Attributes from the graph, nodes, and edges are not copied to the new graph.

4.24 Neighbor degree

4.24.1 Average neighbor degree

average_neighbor_degree(G[, nodes, weighted])	Returns the average degree of the neighborhood of each node.
average_neighbor_in_degree(G[, nodes, weighted])	Returns the average degree of the neighborhood of each node.
average_neighbor_out_degree(G[, nodes, weighted])	Returns the average degree of the neighborhood of each node.

networkx.algorithms.neighbor_degree.average_neighbor_degree

networkx.algorithms.neighbor_degree.**average_neighbor_degree**(*G*, *nodes=None*, *weighted=False*)

Returns the average degree of the neighborhood of each node.

The average 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 [R142],

$$k^w_{nn,i} = frac1s_i \sum_{j \in N(i)} w_{ij} k_j$$

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

nodes: list or iterable (optional) :

Compute neighbor connectivity for these nodes. The default is all nodes.

weighted: bool (default=False) :

Compute weighted average nearest neighbors degree.

Returns **d: dict** :

A dictionary keyed by node with average neighbors degree value.

See Also:

average_neighbor_out_degree, average_neighbor_in_degree, average_degree_connectivity

Notes

For directed graphs you can also specify in-degree or out-degree by calling the relevant functions.

References

[R142]

Examples

```
>>> G=nx.path_graph(4)
>>> G.edge[0][1]['weight'] = 5
>>> G.edge[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, weighted=True)
{0: 2.0, 1: 1.1666666666666667, 2: 1.25, 3: 2.0}

>>> G=nx.DiGraph()
>>> G.add_path([0,1,2,3])
>>> nx.average_neighbor_in_degree(G)
{0: 1.0, 1: 1.0, 2: 1.0, 3: 0.0}
>>> nx.average_neighbor_out_degree(G)
{0: 1.0, 1: 1.0, 2: 0.0, 3: 0.0}
```

networkx.algorithms.neighbor_degree.average_neighbor_in_degree

networkx.algorithms.neighbor_degree.**average_neighbor_in_degree**(*G*, *nodes=None*, *weighted=False*)

Returns the average degree of the neighborhood of each node.

The average 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 [R143],

$$k^w_{nn,i} = frac1s_i \sum_{j \in N(i)} w_{ij} k_j$$

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

nodes: list or iterable (optional) :

Compute neighbor connectivity for these nodes. The default is all nodes.

weighted: bool (default=False) :

Compute weighted average nearest neighbors degree.

Returns **d: dict** :

A dictionary keyed by node with average neighbors degree value.

See Also:

average_neighbor_out_degree, average_neighbor_in_degree, average_degree_connectivity

Notes

For directed graphs you can also specify in-degree or out-degree by calling the relevant functions.

References

[R143]

Examples

```
>>> G=nx.path_graph(4)
>>> G.edge[0][1]['weight'] = 5
>>> G.edge[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, weighted=True)
{0: 2.0, 1: 1.1666666666666667, 2: 1.25, 3: 2.0}

>>> G=nx.DiGraph()
>>> G.add_path([0,1,2,3])
>>> nx.average_neighbor_in_degree(G)
{0: 1.0, 1: 1.0, 2: 1.0, 3: 0.0}
>>> nx.average_neighbor_out_degree(G)
{0: 1.0, 1: 1.0, 2: 0.0, 3: 0.0}
```

networkx.algorithms.neighbor_degree.average_neighbor_out_degree

networkx.algorithms.neighbor_degree.**average_neighbor_out_degree**(*G*, *nodes=None*, *weighted=False*)

Returns the average degree of the neighborhood of each node.

The average 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 [R144],

$$k^w_{nn,i} = frac1s_i \sum_{j \in N(i)} w_{ij} k_j$$

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

nodes: list or iterable (optional) :

Compute neighbor connectivity for these nodes. The default is all nodes.

weighted: bool (default=False) :

Compute weighted average nearest neighbors degree.

Returns **d: dict** :

A dictionary keyed by node with average neighbors degree value.

See Also:

average_neighbor_out_degree, average_neighbor_in_degree, average_degree_connectivity

Notes

For directed graphs you can also specify in-degree or out-degree by calling the relevant functions.

References

[R144]

Examples

```
>>> G=nx.path_graph(4)
>>> G.edge[0][1]['weight'] = 5
>>> G.edge[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, weighted=True)
{0: 2.0, 1: 1.1666666666666667, 2: 1.25, 3: 2.0}

>>> G=nx.DiGraph()
>>> G.add_path([0,1,2,3])
>>> nx.average_neighbor_in_degree(G)
{0: 1.0, 1: 1.0, 2: 1.0, 3: 0.0}
>>> nx.average_neighbor_out_degree(G)
{0: 1.0, 1: 1.0, 2: 0.0, 3: 0.0}
```

4.24.2 Average degree connectivity

average_degree_connectivity(G[, nodes, weighted])	Compute the average degree connectivity of graph.
average_in_degree_connectivity(G[, nodes, ...])	Compute the average degree connectivity of graph.
average_out_degree_connectivity(G[, nodes, ...])	Compute the average degree connectivity of graph.
k_nearest_neighbors(G[, nodes, weighted])	Compute the average degree connectivity of graph.

networkx.algorithms.neighbor_degree.average_degree_connectivity

networkx.algorithms.neighbor_degree.**average_degree_connectivity**(*G*, *nodes=None*, *weighted=False*)

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 [R140],

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}‘$istheweightoftheedgethatlinks$‘$i$ and j, and $N(i)$ are the neighbors of node i.

Parameters **G** : NetworkX graph

nodes: list or iterable (optional) :

Compute neighbor connectivity for these nodes. The default is all nodes.

weighted: bool (default=False) :

Compute weighted average nearest neighbors degree.

Returns **d: dict** :

A dictionary keyed by degree k with the value of average neighbor degree.

See Also:

`neighbors_average_degree`

Notes

This algorithm is sometimes called “k nearest neighbors’.

References

[R140]

Examples

```
>>> G=nx.path_graph(4)
>>> G.edge[1][2]['weight'] = 3
>>> nx.k_nearest_neighbors(G)
{1: 2.0, 2: 1.5}
>>> nx.k_nearest_neighbors(G, weighted=True)
{1: 2.0, 2: 1.75}
```

networkx.algorithms.neighbor_degree.average_in_degree_connectivity

networkx.algorithms.neighbor_degree.**average_in_degree_connectivity**(*G*, *nodes=None*, *weighted=False*)

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 [R141], 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}‘$istheweightoftheedgethatlinks$‘$i$ and j, and $N(i)$ are the neighbors of node i.

Parameters **G** : NetworkX graph

nodes: list or iterable (optional) :

Compute neighbor connectivity for these nodes. The default is all nodes.

weighted: bool (default=False) :

Compute weighted average nearest neighbors degree.

Returns **d: dict** :

A dictionary keyed by degree k with the value of average neighbor degree.

See Also:

`neighbors_average_degree`

Notes

This algorithm is sometimes called “k nearest neighbors’.

References

[R141]

Examples

```
>>> G=nx.path_graph(4)
>>> G.edge[1][2]['weight'] = 3
>>> nx.k_nearest_neighbors(G)
{1: 2.0, 2: 1.5}
>>> nx.k_nearest_neighbors(G, weighted=True)
{1: 2.0, 2: 1.75}
```

networkx.algorithms.neighbor_degree.average_out_degree_connectivity

networkx.algorithms.neighbor_degree.**average_out_degree_connectivity**(*G*, *nodes=None*, *weighted=False*)

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 [R145], 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}‘$istheweightoftheedgethatlinks$‘$i$ and j, and $N(i)$ are the neighbors of node i.

Parameters **G** : NetworkX graph

nodes: list or iterable (optional) :

Compute neighbor connectivity for these nodes. The default is all nodes.

weighted: bool (default=False) :

Compute weighted average nearest neighbors degree.

Returns **d: dict** :

A dictionary keyed by degree k with the value of average neighbor degree.

See Also:

`neighbors_average_degree`

Notes

This algorithm is sometimes called "k nearest neighbors'.

References

[R145]

Examples

```
>>> G=nx.path_graph(4)
>>> G.edge[1][2]['weight'] = 3
>>> nx.k_nearest_neighbors(G)
{1: 2.0, 2: 1.5}
>>> nx.k_nearest_neighbors(G, weighted=True)
{1: 2.0, 2: 1.75}
```

networkx.algorithms.neighbor_degree.k_nearest_neighbors

networkx.algorithms.neighbor_degree.**k_nearest_neighbors**(*G*, *nodes=None*, *weighted=False*)

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 [R146], 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

nodes: list or iterable (optional) :

Compute neighbor connectivity for these nodes. The default is all nodes.

weighted: bool (default=False) :

Compute weighted average nearest neighbors degree.

Returns **d: dict** :

A dictionary keyed by degree k with the value of average neighbor degree.

See Also:

`neighbors_average_degree`

Notes

This algorithm is sometimes called "k nearest neighbors'.

References

[R146]

Examples

```
>>> G=nx.path_graph(4)
>>> G.edge[1][2]['weight'] = 3
>>> nx.k_nearest_neighbors(G)
{1: 2.0, 2: 1.5}
>>> nx.k_nearest_neighbors(G, weighted=True)
{1: 2.0, 2: 1.75}
```

4.25 Rich Club

`rich_club_coefficient`(G[, normalized, Q])	Return the rich-club coefficient of the graph G.

4.25.1 networkx.algorithms.richclub.rich_club_coefficient

`networkx.algorithms.richclub.`**`rich_club_coefficient`**(*G*, *normalized=True*, *Q=100*)

Return the rich-club coefficient of the graph G.

The rich-club coefficient is the ratio, for every degree k, of the number of actual to the number of potential edges for nodes with degree greater than k:

$$\phi(k) = \frac{2Ek}{Nk(Nk-1)}$$

where Nk is the number of nodes with degree larger than k, and Ek be the number of edges among those nodes.

Parameters **G** : NetworkX graph

normalized : bool (optional)

Normalize using randomized network (see [R147])

Q : float (optional, default=100)

If normalized=True build a random network by performing Q*M double-edge swaps, where M is the number of edges in G, to use as a null-model for normalization.

Returns **rc** : dictionary

A dictionary, keyed by degree, with rich club coefficient values.

Notes

The rich club definition and algorithm are found in [R147]. 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 [R148].

References

[R147], [R148]

Examples

```
>>> G = nx.Graph([(0,1),(0,2),(1,2),(1,3),(1,4),(4,5)])
>>> rc = nx.rich_club_coefficient(G,normalized=False)
>>> rc[0]
0.4
```

4.26 Shortest Paths

Compute the shortest paths and path lengths between nodes in the graph.

These algorithms work with undirected and directed graphs.

For directed graphs the paths can be computed in the reverse order by first flipping the edge orientation using R=G.reverse(copy=False).

shortest_path(G[, source, target, weight])	Compute shortest paths in the graph.
shortest_path_length(G[, source, target, weight])	Compute shortest path lengths in the graph.
average_shortest_path_length(G[, weight])	Return the average shortest path length.

4.26.1 networkx.algorithms.shortest_paths.generic.shortest_path

networkx.algorithms.shortest_paths.generic.**shortest_path**(*G*, *source=None*, *target=None*, *weight=None*)

Compute shortest paths in the graph.

Parameters **G** : NetworkX graph

source : node, optional

Starting node for path. If not specified compute shortest paths for all connected node pairs.

target : node, optional

Ending node for path. If not specified compute shortest paths for every node reachable from the source.

weight : None, True or string, optional (default = None)

If None, every edge has weight/distance/cost 1. If True, use the 'weight' edge attribute as the edge weight. If a string, use this edge attribute as the edge weight. Any edge attribute not present defaults to 1.

Returns **path: list or dictionary** :

If the source and target are both specified return a single list of nodes in a shortest path. If only the source is specified return a dictionary keyed by targets with a list of nodes in a shortest path. If neither the source or target is specified return a dictionary of dictionaries with path[source][target]=[list of nodes in path].

See Also:

`all_pairs_shortest_path`, `all_pairs_dijkstra_path`, `single_source_shortest_path`, `single_source_dijkstra_path`

Notes

There may be more than one shortest path between a source and target. This returns only one of them.

For digraphs this returns a shortest directed path. To find paths in the reverse direction use G.reverse(copy=False) first to flip the edge orientation.

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[4]
[0, 1, 2, 3, 4]
>>> p=nx.shortest_path(G) # source,target not specified
>>> p[0][4]
[0, 1, 2, 3, 4]
```

4.26.2 networkx.algorithms.shortest_paths.generic.shortest_path_length

networkx.algorithms.shortest_paths.generic.**shortest_path_length**(*G*, *source=None*, *target=None*, *weight=None*)

Compute shortest path lengths in the graph.

This function can compute the single source shortest path lengths by specifying only the source or all pairs shortest path lengths by specifying neither the source or target.

Parameters **G** : NetworkX graph

source : node, optional

Starting node for path. If not specified compute shortest path lengths for all connected node pairs.

target : node, optional

Ending node for path. If not specified compute shortest path lengths for every node reachable from the source.

weight : None, True or string, optional (default = None)

If None, every edge has weight/distance/cost 1. If True, use the 'weight' edge attribute as the edge weight. If a string, use this edge attribute as the edge weight. Any edge attribute not present defaults to 1.

Returns **length** : number, or container of numbers

If the source and target are both specified return a single number for the shortest path. If only the source is specified return a dictionary keyed by targets with a the shortest path as keys. If neither the source or target is specified return a dictionary of dictionaries with length[source][target]=value.

Raises **NetworkXNoPath** :

If no path exists between source and target.

See Also:

`all_pairs_shortest_path_length`, `all_pairs_dijkstra_path_length`, `single_source_shortest_path_length`, `single_source_dijkstra_path_length`

Notes

For digraphs this returns the shortest directed path. 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)
>>> print(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) # source,target not specified
>>> p[0][4]
4
```

4.26.3 networkx.algorithms.shortest_paths.generic.average_shortest_path_length

`networkx.algorithms.shortest_paths.generic.`**`average_shortest_path_length`**`(`*G*, *weight=None*`)`

Return the average shortest path length.

The average shortest path length is

$$a = \sum_{s,t \in V} \frac{d(s,t)}{n(n-1)}$$

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.

Parameters **G** : NetworkX graph

weight : None, True or string, optional (default = None)

If None, every edge has weight/distance/cost 1. If True, use the 'weight' edge attribute as the edge weight. If a string, use this edge attribute as the edge weight. Any edge attribute not present defaults to 1.

Raises **NetworkXError:** :

if the graph is not connected.

Examples

```
>>> G=nx.path_graph(5)
>>> print(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 g in nx.connected_component_subgraphs(G): ... print(nx.average_shortest_path_length(g)) 1.0 1.0

4.26.4 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.
all_pairs_shortest_path(G[, cutoff])	Compute shortest paths between all nodes.
all_pairs_shortest_path_length(G[, cutoff])	Compute the shortest path lengths between all nodes in G.
predecessor(G, source[, target, cutoff, ...])	Returns dictionary of predecessors for the path from source to all nodes in G.

networkx.algorithms.shortest_paths.unweighted.single_source_shortest_path

networkx.algorithms.shortest_paths.unweighted.**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]
```

networkx.algorithms.shortest_paths.unweighted.single_source_shortest_path_length

networkx.algorithms.shortest_paths.unweighted.**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** : dictionary

Dictionary of shortest path lengths keyed by target.

See Also:

`shortest_path_length`

Examples

```
>>> G=nx.path_graph(5)
>>> length=nx.single_source_shortest_path_length(G,0)
>>> length[4]
4
>>> print(length)
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4}
```

networkx.algorithms.shortest_paths.unweighted.all_pairs_shortest_path

networkx.algorithms.shortest_paths.unweighted.**all_pairs_shortest_path**(*G*, *cutoff=None*)

Compute shortest paths between all nodes.

Parameters **G** : NetworkX graph

cutoff : integer, optional

Depth to stop the search. Only paths of length <= 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=nx.all_pairs_shortest_path(G)
>>> print(path[0][4])
[0, 1, 2, 3, 4]
```

networkx.algorithms.shortest_paths.unweighted.all_pairs_shortest_path_length

networkx.algorithms.shortest_paths.unweighted.**all_pairs_shortest_path_length**(*G*, *cutoff=None*)

Compute the shortest path lengths between all nodes in G.

Parameters **G** : NetworkX graph

cutoff : integer, optional

depth to stop the search. Only paths of length <= cutoff are returned.

Returns **lengths** : dictionary

Dictionary of shortest path lengths keyed by source and target.

Notes

The dictionary returned only has keys for reachable node pairs.

Examples

```
>>> G=nx.path_graph(5)
>>> length=nx.all_pairs_shortest_path_length(G)
>>> print(length[1][4])
3
>>> length[1]
{0: 1, 1: 0, 2: 1, 3: 2, 4: 3}
```

networkx.algorithms.shortest_paths.unweighted.predecessor

networkx.algorithms.shortest_paths.unweighted.**predecessor**(*G*, *source*, *target=None*, *cutoff=None*, *return_seen=None*)

Returns dictionary 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.

Returns **pred** : dictionary

Dictionary, keyed by node, of predecessors in the shortest path.

Examples

```
>>> G=nx.path_graph(4)
>>> print(G.nodes())
[0, 1, 2, 3]
>>> nx.predecessor(G,0)
{0: [], 1: [0], 2: [1], 3: [2]}
```

Shortest path algorithms for weighed graphs.

dijkstra_path(G, source, target[, weight])	Returns the shortest path from source to target in a weighted graph G.
dijkstra_path_length(G, source, target[, weight])	Returns the shortest path length from source to target in a weighted graph.
single_source_dijkstra_path(G, source[, ...])	Compute shortest path between source and all other reachable nodes for a weighted graph.
single_source_dijkstra_path_length(G, source)	Compute the shortest path length between source and all other reachable nodes for a weighted graph.
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.
single_source_dijkstra(G, source[, target, ...])	Compute shortest paths and lengths in a weighted graph G.
bidirectional_dijkstra(G, source, target[, ...])	Dijkstra's algorithm for shortest paths using bidirectional search.
dijkstra_predecessor_and_distance(G, source)	Compute shortest path length and predecessors on shortest paths in weighted graphs.
bellman_ford(G, source[, weight])	Compute shortest path lengths and predecessors on shortest paths in weighted graphs.
negative_edge_cycle(G[, weight])	Return True if there exists a negative edge cycle anywhere in G.

networkx.algorithms.shortest_paths.weighted.dijkstra_path

networkx.algorithms.shortest_paths.weighted.**dijkstra_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, optional (default='weight') :

Edge data key corresponding to the edge weight

Returns **path** : list

List of nodes in a shortest path.

Raises **NetworkXNoPath** :

If no path exists between source and target.

See Also:

bidirectional_dijkstra

Notes

Edge weight attributes must be numerical. Distances are calculated as sums of weighted edges traversed.

Examples

```
>>> G=nx.path_graph(5)
>>> print(nx.dijkstra_path(G,0,4))
[0, 1, 2, 3, 4]
```

networkx.algorithms.shortest_paths.weighted.dijkstra_path_length

networkx.algorithms.shortest_paths.weighted.**dijkstra_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, optional (default='weight') :

Edge data key corresponding to the edge weight

Returns **length** : number

Shortest path length.

Raises **NetworkXNoPath** :

If no path exists between source and target.

See Also:

`bidirectional_dijkstra`

Notes

Edge weight attributes must be numerical. Distances are calculated as sums of weighted edges traversed.

Examples

```
>>> G=nx.path_graph(5)
>>> print(nx.dijkstra_path_length(G,0,4))
4
```

networkx.algorithms.shortest_paths.weighted.single_source_dijkstra_path

networkx.algorithms.shortest_paths.weighted.**single_source_dijkstra_path**(*G*, *source*, *cutoff=None*, *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, optional (default='weight') :

Edge data key corresponding to the edge weight

cutoff : integer or float, optional

Depth to stop the search. Only paths of length <= cutoff are returned.

Returns **paths** : dictionary

Dictionary of shortest path lengths keyed by target.

See Also:

`single_source_dijkstra`

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_dijkstra_path(G,0)
>>> path[4]
[0, 1, 2, 3, 4]
```

networkx.algorithms.shortest_paths.weighted.single_source_dijkstra_path_length

networkx.algorithms.shortest_paths.weighted.**single_source_dijkstra_path_length**(*G*, *source*, *cutoff=None*, *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, optional (default='weight') :

Edge data key corresponding to the edge weight.

cutoff : integer or float, optional

Depth to stop the search. Only paths of length <= cutoff are returned.

Returns **length** : dictionary

Dictionary of shortest lengths keyed by target.

See Also:

single_source_dijkstra

Notes

Edge weight attributes must be numerical. Distances are calculated as sums of weighted edges traversed.

Examples

```
>>> G=nx.path_graph(5)
>>> length=nx.single_source_dijkstra_path_length(G,0)
>>> length[4]
4
>>> print(length)
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4}
```

networkx.algorithms.shortest_paths.weighted.all_pairs_dijkstra_path

networkx.algorithms.shortest_paths.weighted.**all_pairs_dijkstra_path**(*G*, *cutoff=None*, *weight='weight'*)

Compute shortest paths between all nodes in a weighted graph.

Parameters **G** : NetworkX graph

weight: string, optional (default='weight') :

Edge data key corresponding to the edge weight

cutoff : integer or float, optional

Depth to stop the search. Only paths of length <= cutoff are returned.

Returns **distance** : dictionary

Dictionary, keyed by source and target, of shortest paths.

See Also:

`floyd_warshall`

Notes

Edge weight attributes must be numerical. Distances are calculated as sums of weighted edges traversed.

Examples

```
>>> G=nx.path_graph(5)
>>> path=nx.all_pairs_dijkstra_path(G)
>>> print(path[0][4])
[0, 1, 2, 3, 4]
```

networkx.algorithms.shortest_paths.weighted.all_pairs_dijkstra_path_length

networkx.algorithms.shortest_paths.weighted.**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

weight: string, optional (default='weight') :

Edge data key corresponding to the edge weight

cutoff : integer or float, optional

Depth to stop the search. Only paths of length <= cutoff are returned.

Returns **distance** : dictionary

Dictionary, keyed by source and target, of shortest path lengths.

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=nx.all_pairs_dijkstra_path_length(G)
>>> print(length[1][4])
3
>>> length[1]
{0: 1, 1: 0, 2: 1, 3: 2, 4: 3}
```

networkx.algorithms.shortest_paths.weighted.single_source_dijkstra

networkx.algorithms.shortest_paths.weighted.**single_source_dijkstra**(*G*, *source*, *target=None*, *cutoff=None*, *weight='weight'*)

Compute shortest paths and lengths in a weighted graph G.

Uses Dijkstra's algorithm for shortest paths.

Parameters **G** : NetworkX graph

source : node label

Starting node for path

target : node label, optional

Ending node for path

cutoff : integer or float, optional

Depth to stop the search. Only paths of length <= cutoff are returned.

Returns **distance,path** : dictionaries

Returns a tuple of two dictionaries keyed by node. The first dictionary stores distance from the source. The second stores the path from the source to that node.

See Also:

single_source_dijkstra_path, single_source_dijkstra_path_length

Notes

Edge weight attributes must be numerical. Distances are calculated as sums of weighted edges traversed.

Based on the Python cookbook recipe (119466) at http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/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)
>>> print(length[4])
4
>>> print(length)
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4}
>>> path[4]
[0, 1, 2, 3, 4]
```

networkx.algorithms.shortest_paths.weighted.bidirectional_dijkstra

networkx.algorithms.shortest_paths.weighted.**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, optional (default='weight') :

Edge data key corresponding to the edge weight

Returns **length** : number

Shortest path length.

Returns a tuple of two dictionaries keyed by node. :

The first dictionary stores distance from the source. :

The second stores the path from the source to that node. :

Raises **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.

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]
```

networkx.algorithms.shortest_paths.weighted.dijkstra_predecessor_and_distance

networkx.algorithms.shortest_paths.weighted.**dijkstra_predecessor_and_distance**(*G*, *source*, *cutoff=None*, *weight='weight'*)

Compute shortest path length and predecessors on shortest paths in weighted graphs.

Parameters **G** : NetworkX graph

source : node label

Starting node for path

weight: string, optional (default='weight') :

Edge data key corresponding to the edge weight

cutoff : integer or float, optional

Depth to stop the search. Only paths of length <= cutoff are returned.

Returns **pred,distance** : dictionaries

Returns two dictionaries representing a list of predecessors of a node and the distance to each node.

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.

networkx.algorithms.shortest_paths.weighted.bellman_ford

networkx.algorithms.shortest_paths.weighted.**bellman_ford**(*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, optional (default='weight') :

Edge data key corresponding to the edge weight

Returns **pred, dist** : dictionaries

Returns two dictionaries keyed by node to predecessor in the path and to the distance from the source respectively.

Raises **NetworkXUnbounded** :

If the (di)graph contains a negative cost (di)cycle, the algorithm raises an exception to indicate the presence of the negative cost (di)cycle. Note: any negative weight edge in an undirected graph is a negative cost 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 cost (di)cycle, it will not be detected.

Examples

```
>>> import networkx as nx
>>> G = nx.path_graph(5, create_using = nx.DiGraph())
>>> pred, dist = nx.bellman_ford(G, 0)
>>> pred
{0: None, 1: 0, 2: 1, 3: 2, 4: 3}
>>> dist
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4}

>>> from nose.tools import assert_raises
>>> G = nx.cycle_graph(5, create_using = nx.DiGraph())
>>> G[1][2]['weight'] = -7
>>> assert_raises(nx.NetworkXUnbounded, nx.bellman_ford, G, 0)
```

networkx.algorithms.shortest_paths.weighted.negative_edge_cycle

networkx.algorithms.shortest_paths.weighted.**negative_edge_cycle**(*G*, *weight='weight'*)

Return True if there exists a negative edge cycle anywhere in G.

Parameters **G** : NetworkX graph

weight: string, optional (default='weight') :

Edge data key corresponding to the edge weight

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() but finds negative cycles on any component by first adding a new node connected to every node, and starting bellman_ford on that node. It then removes that extra node.

Examples

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

4.26.5 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.

networkx.algorithms.shortest_paths.dense.floyd_warshall

networkx.algorithms.shortest_paths.dense.**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 is O(n^2).

networkx.algorithms.shortest_paths.dense.floyd_warshall_predecessor_and_distance

networkx.algorithms.shortest_paths.dense.**floyd_warshall_predecessor_and_distance**(*G*, *weight='weig*

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 is O(n^2).

networkx.algorithms.shortest_paths.dense.floyd_warshall_numpy

networkx.algorithms.shortest_paths.dense.**floyd_warshall_numpy**(*G*, *nodelist=None*, *weight='weight'*)

Find all-pairs shortest path lengths using Floyd's algorithm.

Parameters **G** : NetworkX graph

nodelist : list, optional

The rows and columns are ordered by the nodes in nodelist. If nodelist is None then the ordering is produced by G.nodes().

weight: string, optional (default= 'weight') :

Edge data key corresponding to the edge weight.

Returns **distance** : NumPy matrix

A matrix of shortest path distances between nodes. If there is no path between to nodes the corresponding matrix entry will be Inf.

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 is O(n^2).

4.26.6 A* Algorithm

Shortest paths and path lengths using A* ("A star") algorithm.

astar_path(G, source, target[, heuristic, ...])	Return a list of nodes in a shortest path between source and target
astar_path_length(G, source, target[, ...])	Return a list of nodes in a shortest path between source and target

networkx.algorithms.shortest_paths.astar.astar_path

networkx.algorithms.shortest_paths.astar.**astar_path**(*G*, *source*, *target*, *heuristic=None*, *weight='weight'*)

Return 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.

weight: string, optional (default='weight') :

Edge data key corresponding to the edge weight.

Raises **NetworkXNoPath** :

If no path exists between source and target.

See Also:

shortest_path, dijkstra_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)
>>> 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),dist))
[(0, 0), (0, 1), (1, 1), (1, 2), (2, 2)]
```

networkx.algorithms.shortest_paths.astar.astar_path_length

networkx.algorithms.shortest_paths.astar.**astar_path_length**(*G*, *source*, *target*, *heuristic=None*, *weight='weight'*)

Return a list of nodes in a 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.

Raises **NetworkXNoPath** :

If no path exists between source and target.

See Also:

astar_path

4.27 Traversal

4.27.1 Depth First Search

Basic algorithms for depth-first searching.

Based on http://www.ics.uci.edu/~eppstein/PADS/DFS.py by D. Eppstein, July 2004.

dfs_edges(G[, source])	Produce edges in a depth-first-search starting at source.
dfs_tree(G[, source])	Return directed tree of depth-first-search from source.
dfs_predecessors(G[, source])	Return dictionary of predecessors in depth-first-search from source.
dfs_successors(G[, source])	Return dictionary of successors in depth-first-search from source.
dfs_preorder_nodes(G[, source])	Produce nodes in a depth-first-search pre-ordering starting at source.
dfs_postorder_nodes(G[, source])	Produce nodes in a depth-first-search post-ordering starting
dfs_labeled_edges(G[, source])	Produce edges in a depth-first-search starting at source and

networkx.algorithms.traversal.depth_first_search.dfs_edges

networkx.algorithms.traversal.depth_first_search.**dfs_edges**(*G*, *source=None*)

Produce edges in a depth-first-search starting at source.

networkx.algorithms.traversal.depth_first_search.dfs_tree

networkx.algorithms.traversal.depth_first_search.**dfs_tree**(*G*, *source=None*)

Return directed tree of depth-first-search from source.

networkx.algorithms.traversal.depth_first_search.dfs_predecessors

networkx.algorithms.traversal.depth_first_search.**dfs_predecessors**(*G*, *source=None*)

Return dictionary of predecessors in depth-first-search from source.

networkx.algorithms.traversal.depth_first_search.dfs_successors

networkx.algorithms.traversal.depth_first_search.**dfs_successors**(*G*, *source=None*)

Return dictionary of successors in depth-first-search from source.

networkx.algorithms.traversal.depth_first_search.dfs_preorder_nodes

networkx.algorithms.traversal.depth_first_search.**dfs_preorder_nodes**(*G*, *source=None*)

Produce nodes in a depth-first-search pre-ordering starting at source.

networkx.algorithms.traversal.depth_first_search.dfs_postorder_nodes

networkx.algorithms.traversal.depth_first_search.**dfs_postorder_nodes**(*G*, *source=None*)

Produce nodes in a depth-first-search post-ordering starting from source.

networkx.algorithms.traversal.depth_first_search.dfs_labeled_edges

networkx.algorithms.traversal.depth_first_search.**dfs_labeled_edges**(*G*, *source=None*)

Produce edges in a depth-first-search starting at source and labeled by direction type (forward, reverse, nontree).

4.27.2 Breadth First Search

Basic algorithms for breadth-first searching.

bfs_edges(G, source)	Produce edges in a breadth-first-search starting at source.
bfs_tree(G, source)	Return directed tree of breadth-first-search from source.
bfs_predecessors(G, source)	Return dictionary of predecessors in breadth-first-search from source.
bfs_successors(G, source)	Return dictionary of successors in breadth-first-search from source.

networkx.algorithms.traversal.breadth_first_search.bfs_edges

networkx.algorithms.traversal.breadth_first_search.**bfs_edges**(*G*, *source*)

Produce edges in a breadth-first-search starting at source.

networkx.algorithms.traversal.breadth_first_search.bfs_tree

networkx.algorithms.traversal.breadth_first_search.**bfs_tree**(*G*, *source*)

Return directed tree of breadth-first-search from source.

networkx.algorithms.traversal.breadth_first_search.bfs_predecessors

networkx.algorithms.traversal.breadth_first_search.**bfs_predecessors**(*G*, *source*)

Return dictionary of predecessors in breadth-first-search from source.

networkx.algorithms.traversal.breadth_first_search.bfs_successors

networkx.algorithms.traversal.breadth_first_search.**bfs_successors**(*G*, *source*)

Return dictionary of successors in breadth-first-search from source.

4.28 Vitality

Vitality measures.

closeness_vitality(G[, v, weight])	Compute closeness vitality for nodes.

4.28.1 networkx.algorithms.vitality.closeness_vitality

networkx.algorithms.vitality.**closeness_vitality**(*G*, *v=None*, *weight=None*)

Compute closeness vitality for nodes.

Closeness vitality at a node is the change in the sum of distances between all node pairs when excluding a that node.

Parameters **G** : graph

A networkx graph

v : node, optional

Return only the value for node v.

weight : None, True or string, optional

If None, edge weights are ignored. If True, edge attribute 'weight' is used as weight of each edge. Otherwise holds the name of the edge attribute used as weight.

Returns **nodes** : dictionary

Dictionary with nodes as keys and closeness vitality as the value.

See Also:

closeness_centrality

Examples

```
>>> G=nx.cycle_graph(3)
>>> nx.closeness_vitality(G)
{0: 4.0, 1: 4.0, 2: 4.0}
```

CHAPTER

FIVE

FUNCTIONS

Functional interface to graph methods and assorted utilities.

5.1 Graph

degree(G[, nbunch, weighted])	Return degree of single node or of nbunch of nodes.
degree_histogram(G)	Return a list of the frequency of each degree value.
density(G)	Return the density of a graph.
info(G[, n])	Print short summary of information for the graph G or the node n.
create_empty_copy(G[, with_nodes])	Return a copy of the graph G with all of the edges removed.
is_directed(G)	Return True if graph is directed.

5.1.1 networkx.classes.function.degree

networkx.classes.function.**degree**(*G*, *nbunch=None*, *weighted=False*)

Return degree of single node or of nbunch of nodes. If nbunch is ommitted, then return degrees of *all* nodes.

5.1.2 networkx.classes.function.degree_histogram

networkx.classes.function.**degree_histogram**(*G*)

Return 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))

5.1.3 networkx.classes.function.density

networkx.classes.function.**density**(*G*)

Return 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 an graph without edges and 1.0 for a complete graph.

The density of multigraphs can be higher than 1.

5.1.4 networkx.classes.function.info

networkx.classes.function.**info**(*G*, *n=None*)

Print short summary of information for the graph G or the node n.

Parameters **G** : Networkx graph

A graph

n : node (any hashable)

A node in the graph G

5.1.5 networkx.classes.function.create_empty_copy

networkx.classes.function.**create_empty_copy**(*G*, *with_nodes=True*)

Return a copy of the graph G with all of the edges removed.

Parameters **G** : graph

A NetworkX graph

with_nodes : bool (default=True)

Include nodes.

Notes

Graph, node, and edge data is not propagated to the new graph.

5.1.6 networkx.classes.function.is_directed

networkx.classes.function.**is_directed**(*G*)

Return True if graph is directed.

5.2 Nodes

nodes(G)	Return a copy of the graph nodes in a list.
number_of_nodes(G)	Return the number of nodes in the graph.
nodes_iter(G)	Return an iterator over the graph nodes.

5.2.1 networkx.classes.function.nodes

networkx.classes.function.**nodes**(*G*)
Return a copy of the graph nodes in a list.

5.2.2 networkx.classes.function.number_of_nodes

networkx.classes.function.**number_of_nodes**(*G*)
Return the number of nodes in the graph.

5.2.3 networkx.classes.function.nodes_iter

networkx.classes.function.**nodes_iter**(*G*)
Return an iterator over the graph nodes.

5.3 Edges

edges(G[, nbunch])	Return list of edges adjacent to nodes in nbunch.
number_of_edges(G)	Return the number of edges in the graph.
edges_iter(G[, nbunch])	Return iterator over edges adjacent to nodes in nbunch.

5.3.1 networkx.classes.function.edges

networkx.classes.function.**edges**(*G*, *nbunch=None*)
Return list of edges adjacent to nodes in nbunch.

Return all edges if nbunch is unspecified or nbunch=None.

For digraphs, edges=out_edges

5.3.2 networkx.classes.function.number_of_edges

networkx.classes.function.**number_of_edges**(*G*)
Return the number of edges in the graph.

5.3.3 networkx.classes.function.edges_iter

networkx.classes.function.**edges_iter**(*G*, *nbunch=None*)
Return iterator over edges adjacent to nodes in nbunch.

Return all edges if nbunch is unspecified or nbunch=None.

For digraphs, edges=out_edges

5.4 Attributes

set_node_attributes(G, name, attributes)	Set node attributes from dictionary of nodes and values
get_node_attributes(G, name)	Get node attributes from graph
set_edge_attributes(G, name, attributes)	Set edge attributes from dictionary of edge tuples and values
get_edge_attributes(G, name)	Get edge attributes from graph

5.4.1 networkx.classes.function.set_node_attributes

networkx.classes.function.**set_node_attributes**(*G*, *name*, *attributes*)

Set node attributes from dictionary of nodes and values

Parameters **G** : NetworkX Graph

name : string

Attribute name

attributes: dict :

Dictionary of attributes keyed by node.

Examples

```
>>> G=nx.path_graph(3)
>>> bb=nx.betweenness_centrality(G)
>>> nx.set_node_attributes(G,'betweenness',bb)
>>> G.node[1]['betweenness']
1.0
```

5.4.2 networkx.classes.function.get_node_attributes

networkx.classes.function.**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'
```

5.4.3 networkx.classes.function.set_edge_attributes

networkx.classes.function.**set_edge_attributes**(*G*, *name*, *attributes*)

Set edge attributes from dictionary of edge tuples and values

Parameters **G** : NetworkX Graph

name : string

Attribute name

attributes: dict :

Dictionary of attributes keyed by edge (tuple).

Examples

```
>>> G=nx.path_graph(3)
>>> bb=nx.edge_betweenness_centrality(G)
>>> nx.set_edge_attributes(G,'betweenness',bb)
>>> G[1][2]['betweenness']
4.0
```

5.4.4 networkx.classes.function.get_edge_attributes

networkx.classes.function.**get_edge_attributes**(*G*, *name*)

Get edge attributes from graph

Parameters **G** : NetworkX Graph

name : string

Attribute name

Returns **Dictionary of attributes keyed by node.** :

Examples

```
>>> G=nx.Graph()
>>> G.add_path([1,2,3],color='red')
>>> color=nx.get_edge_attributes(G,'color')
>>> color[(1,2)]
'red'
```

5.5 Freezing graph structure

freeze(G)	Modify graph to prevent addition of nodes or edges.
is_frozen(G)	Return True if graph is frozen.

5.5.1 networkx.classes.function.freeze

networkx.classes.function.**freeze**(*G*)

Modify graph to prevent addition of nodes or edges.

Parameters **G** : graph

A NetworkX graph

See Also:

`is_frozen`

Notes

This does not prevent modification of edge data.

To "unfreeze" a graph you must make a copy.

Examples

```
>>> G=nx.Graph()
>>> G.add_path([0,1,2,3])
>>> G=nx.freeze(G)
>>> try:
...    G.add_edge(4,5)
... except nx.NetworkXError as e:
...    print(str(e))
Frozen graph can't be modified
```

5.5.2 networkx.classes.function.is_frozen

networkx.classes.function.**is_frozen**(*G*)

Return True if graph is frozen.

Parameters **G** : graph

A NetworkX graph

See Also:

`freeze`

CHAPTER

SIX

GRAPH GENERATORS

6.1 Atlas

Generators for the small graph atlas.

See "An Atlas of Graphs" by Ronald C. Read and Robin J. Wilson, Oxford University Press, 1998.

Because of its size, this module is not imported by default.

graph_atlas_g()	Return the list [G0,G1,...,G1252] of graphs as named in the Graph Atlas.

6.1.1 networkx.generators.atlas.graph_atlas_g

networkx.generators.atlas.**graph_atlas_g**()

Return the list [G0,G1,...,G1252] of graphs as named in the Graph Atlas. G0,G1,...,G1252 are all graphs with up to 7 nodes.

The graphs are listed:

1. in increasing order of number of nodes;
2. for a fixed number of nodes, in increasing order of the number of edges;
3. for fixed numbers of nodes and edges, in increasing order of the degree sequence, for example 111223 < 112222;
4. for fixed degree sequence, in increasing number of automorphisms.

Note that indexing is set up so that for GAG=graph_atlas_g(), then G123=GAG[123] and G[0]=empty_graph(0)

6.2 Classic

Generators for some classic graphs.

The typical graph generator 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 generators in this module return a Graph class (i.e. a simple, undirected graph).

balanced_tree(r, h[, create_using])	Return the perfectly balanced r-tree of height h.
barbell_graph(m1, m2[, create_using])	Return the Barbell Graph: two complete graphs connected by a path.
complete_graph(n[, create_using])	Return the complete graph K_n with n nodes.
complete_bipartite_graph(n1, n2[, create_using])	Return the complete bipartite graph K_{n1_n2}.
circular_ladder_graph(n[, create_using])	Return the circular ladder graph CL_n of length n.
cycle_graph(n[, create_using])	Return the cycle graph C_n over n nodes.
dorogovtsev_goltsev_mendes_graph(n[, ...])	Return the hierarchically constructed Dorogovtsev-Goltsev-Mendes graph.
empty_graph([n, create_using])	Return the empty graph with n nodes and zero edges.
grid_2d_graph(m, n[, periodic, create_using])	Return the 2d grid graph of mxn nodes, each connected to its nearest neighbors.
grid_graph(dim[, periodic, create_using])	Return the n-dimensional grid graph.
hypercube_graph(n[, create_using])	Return the n-dimensional hypercube.
ladder_graph(n[, create_using])	Return the Ladder graph of length n.
lollipop_graph(m, n[, create_using])	Return the Lollipop Graph; K_m connected to P_n.
null_graph([create_using])	Return the Null graph with no nodes or edges.
path_graph(n[, create_using])	Return the Path graph P_n of n nodes linearly connected by n-1 edges.
star_graph(n[, create_using])	Return the Star graph with n+1 nodes: one center node, connected to n outer nodes.
trivial_graph([create_using])	Return the Trivial graph with one node (with integer label 0) and no edges.
wheel_graph(n[, create_using])	Return the wheel graph: a single hub node connected to each node of the (n-1)-node cycle graph.

6.2.1 networkx.generators.classic.balanced_tree

networkx.generators.classic.**balanced_tree**(*r*, *h*, *create_using=None*)

Return the perfectly balanced r-tree of height h.

Parameters **r** : int

Branching factor of the tree

h : int

Height of the tree

create_using : NetworkX graph type, optional

Use specified type to construct graph (default = networkx.Graph)

Returns **G** : networkx Graph

A tree with n nodes

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 the integers 0 (the root) up to number_of_nodes - 1.

Also refered to as a complete r-ary tree.

6.2.2 networkx.generators.classic.barbell_graph

networkx.generators.classic.**barbell_graph**(*m1*, *m2*, *create_using=None*)

Return the Barbell Graph: two complete graphs connected by a path.

For m1 > 1 and m2 >= 0.

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 etext on Random Walks on Graphs.

6.2.3 networkx.generators.classic.complete_graph

networkx.generators.classic.**complete_graph**(*n*, *create_using=None*)

Return the complete graph K_n with n nodes.

Node labels are the integers 0 to n-1.

6.2.4 networkx.generators.classic.complete_bipartite_graph

networkx.generators.classic.**complete_bipartite_graph**(*n1*, *n2*, *create_using=None*)

Return the complete bipartite graph K_{n1_n2}.

Composed of two partitions with n1 nodes in the first and n2 nodes in the second. Each node in the first is connected to each node in the second.

Node labels are the integers 0 to n1+n2-1

6.2.5 networkx.generators.classic.circular_ladder_graph

networkx.generators.classic.**circular_ladder_graph**(*n*, *create_using=None*)

Return 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

6.2.6 networkx.generators.classic.cycle_graph

networkx.generators.classic.**cycle_graph**(*n*, *create_using=None*)

Return the cycle graph C_n over n nodes.

C_n is the n-path with two end-nodes connected.

Node labels are the integers 0 to n-1 If create_using is a DiGraph, the direction is in increasing order.

6.2.7 networkx.generators.classic.dorogovtsev_goltsev_mendes_graph

networkx.generators.classic.**dorogovtsev_goltsev_mendes_graph**(*n*, *create_using=None*)

Return the hierarchically constructed Dorogovtsev-Goltsev-Mendes graph.

n is the generation. See: arXiv:/cond-mat/0112143 by Dorogovtsev, Goltsev and Mendes.

6.2.8 networkx.generators.classic.empty_graph

networkx.generators.classic.**empty_graph**(*n=0*, *create_using=None*)

Return the empty graph with n nodes and zero edges.

Node labels are the integers 0 to n-1

For example: >>> G=nx.empty_graph(10) >>> G.number_of_nodes() 10 >>> G.number_of_edges() 0

The variable create_using should point to a "graph"-like object that will be cleaned (nodes and edges will be removed) and refitted as an empty "graph" with n nodes with integer labels. 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 two main uses: Firstly, the variable create_using can be used to create an empty digraph, network,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, pseudograph, etc.) via create_using. For example, if G is an existing graph (resp. digraph, pseudograph, etc.), then empty_graph(n,create_using=G) will empty G (i.e. delete all nodes and edges using G.clear() in base) and then add n nodes and zero edges, and return the modified graph (resp. digraph, pseudograph, etc.).

See also create_empty_copy(G).

6.2.9 networkx.generators.classic.grid_2d_graph

networkx.generators.classic.**grid_2d_graph**(*m*, *n*, *periodic=False*, *create_using=None*)

Return the 2d grid graph of mxn nodes, each connected to its nearest neighbors. Optional argument periodic=True will connect boundary nodes via periodic boundary conditions.

6.2.10 networkx.generators.classic.grid_graph

networkx.generators.classic.**grid_graph**(*dim*, *periodic=False*, *create_using=None*)

Return the n-dimensional grid graph.

The dimension is the length of the list 'dim' and the size in each dimension is the value of the list element.

E.g. G=grid_graph(dim=[2,3]) produces a 2x3 grid graph.

If periodic=True then join grid edges with periodic boundary conditions.

6.2.11 networkx.generators.classic.hypercube_graph

networkx.generators.classic.**hypercube_graph**(*n*, *create_using=None*)
Return the n-dimensional hypercube.

Node labels are the integers 0 to 2**n - 1.

6.2.12 networkx.generators.classic.ladder_graph

networkx.generators.classic.**ladder_graph**(*n*, *create_using=None*)
Return the Ladder graph of length n.

This is two rows of n nodes, with each pair connected by a single edge.

Node labels are the integers 0 to 2*n - 1.

6.2.13 networkx.generators.classic.lollipop_graph

networkx.generators.classic.**lollipop_graph**(*m*, *n*, *create_using=None*)
Return the Lollipop Graph; K_m connected to P_n.

This is the Barbell Graph without the right barbell.

For m>1 and n>=0, the complete graph K_m is connected to the path P_n. The resulting m+n nodes are labelled 0,...,m-1 for the complete graph and m,...,m+n-1 for the path. The 2 subgraphs are joined via the edge (m-1,m). If n=0, this is merely a complete graph.

Node labels are the integers 0 to number_of_nodes - 1.

(This graph is an extremal example in David Aldous and Jim Fill's etext on Random Walks on Graphs.)

6.2.14 networkx.generators.classic.null_graph

networkx.generators.classic.**null_graph**(*create_using=None*)
Return the Null graph with no nodes or edges.

See empty_graph for the use of create_using.

6.2.15 networkx.generators.classic.path_graph

networkx.generators.classic.**path_graph**(*n*, *create_using=None*)
Return the Path graph P_n of n nodes linearly connected by n-1 edges.

Node labels are the integers 0 to n - 1. If create_using is a DiGraph then the edges are directed in increasing order.

6.2.16 networkx.generators.classic.star_graph

networkx.generators.classic.**star_graph**(*n*, *create_using=None*)
Return the Star graph with n+1 nodes: one center node, connected to n outer nodes.

Node labels are the integers 0 to n.

6.2.17 networkx.generators.classic.trivial_graph

networkx.generators.classic.**trivial_graph**(*create_using=None*)
Return the Trivial graph with one node (with integer label 0) and no edges.

6.2.18 networkx.generators.classic.wheel_graph

networkx.generators.classic.**wheel_graph**(*n*, *create_using=None*)
Return the wheel graph: a single hub node connected to each node of the (n-1)-node cycle graph.

Node labels are the integers 0 to n - 1.

6.3 Small

Various small and named graphs, together with some compact generators.

make_small_graph(graph_description[, ...])	Return the small graph described by graph_description.
LCF_graph(n, shift_list, repeats[, create_using])	Return the cubic graph specified in LCF notation.
bull_graph([create_using])	Return the Bull graph.
chvatal_graph([create_using])	Return the Chvátal graph.
cubical_graph([create_using])	Return the 3-regular Platonic Cubical graph.
desargues_graph([create_using])	Return the Desargues graph.
diamond_graph([create_using])	Return the Diamond graph.
dodecahedral_graph([create_using])	Return the Platonic Dodecahedral graph.
frucht_graph([create_using])	Return the Frucht Graph.
heawood_graph([create_using])	Return the Heawood graph, a (3,6) cage.
house_graph([create_using])	Return the House graph (square with triangle on top).
house_x_graph([create_using])	Return the House graph with a cross inside the house square.
icosahedral_graph([create_using])	Return the Platonic Icosahedral graph.
krackhardt_kite_graph([create_using])	Return the Krackhardt Kite Social Network.
moebius_kantor_graph([create_using])	Return the Moebius-Kantor graph.
octahedral_graph([create_using])	Return the Platonic Octahedral graph.
pappus_graph()	Return the Pappus graph.
petersen_graph([create_using])	Return the Petersen graph.
sedgewick_maze_graph([create_using])	Return a small maze with a cycle.
tetrahedral_graph([create_using])	Return the 3-regular Platonic Tetrahedral graph.
truncated_cube_graph([create_using])	Return the skeleton of the truncated cube.
truncated_tetrahedron_graph([create_using])	Return the skeleton of the truncated Platonic tetrahedron.
tutte_graph([create_using])	Return the Tutte graph.

6.3.1 networkx.generators.small.make_small_graph

networkx.generators.small.**make_small_graph**(*graph_description*, *create_using=None*)
Return the small graph described by graph_description.

graph_description is a list of the form [ltype,name,n,xlist]

Here ltype is one of “adjacencylist” or “edgelist”, name is the name of the graph and n the number of nodes. This constructs a graph of n nodes with integer labels 0,..,n-1.

If ltype=”adjacencylist” then xlist is an adjacency list with exactly n entries, in with the j’th entry (which can be empty) specifies the nodes connected to vertex j. e.g. the “square” graph C_4 can be obtained by

```
>>> G=nx.make_small_graph(["adjacencylist","C_4",4,[[2,4],[1,3],[2,4],[1,3]]])
```

or, since we do not need to add edges twice,

```
>>> G=nx.make_small_graph(["adjacencylist","C_4",4,[[2,4],[3],[4],[]]])
```

If ltype="edgelist" then xlist is an edge list written as [[v1,w2],[v2,w2],...,[vk,wk]], where vj and wj integers in the range 1,..,n e.g. the "square" graph C_4 can be obtained by

```
>>> G=nx.make_small_graph(["edgelist","C_4",4,[[1,2],[3,4],[2,3],[4,1]]])
```

Use the create_using argument to choose the graph class/type.

6.3.2 networkx.generators.small.LCF_graph

networkx.generators.small.**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.

6.3.3 networkx.generators.small.bull_graph

networkx.generators.small.**bull_graph**(*create_using=None*)

Return the Bull graph.

6.3.4 networkx.generators.small.chvatal_graph

networkx.generators.small.**chvatal_graph**(*create_using=None*)

Return the Chvátal graph.

6.3.5 networkx.generators.small.cubical_graph

networkx.generators.small.**cubical_graph**(*create_using=None*)
Return the 3-regular Platonic Cubical graph.

6.3.6 networkx.generators.small.desargues_graph

networkx.generators.small.**desargues_graph**(*create_using=None*)
Return the Desargues graph.

6.3.7 networkx.generators.small.diamond_graph

networkx.generators.small.**diamond_graph**(*create_using=None*)
Return the Diamond graph.

6.3.8 networkx.generators.small.dodecahedral_graph

networkx.generators.small.**dodecahedral_graph**(*create_using=None*)
Return the Platonic Dodecahedral graph.

6.3.9 networkx.generators.small.frucht_graph

networkx.generators.small.**frucht_graph**(*create_using=None*)
Return the Frucht Graph.

The Frucht Graph is the smallest cubical graph whose automorphism group consists only of the identity element.

6.3.10 networkx.generators.small.heawood_graph

networkx.generators.small.**heawood_graph**(*create_using=None*)
Return the Heawood graph, a (3,6) cage.

6.3.11 networkx.generators.small.house_graph

networkx.generators.small.**house_graph**(*create_using=None*)
Return the House graph (square with triangle on top).

6.3.12 networkx.generators.small.house_x_graph

networkx.generators.small.**house_x_graph**(*create_using=None*)
Return the House graph with a cross inside the house square.

6.3.13 networkx.generators.small.icosahedral_graph

networkx.generators.small.**icosahedral_graph**(*create_using=None*)
Return the Platonic Icosahedral graph.

6.3.14 networkx.generators.small.krackhardt_kite_graph

networkx.generators.small.**krackhardt_kite_graph**(*create_using=None*)
Return the Krackhardt Kite Social Network.

A 10 actor social network introduced by David Krackhardt to illustrate: degree, betweenness, centrality, closeness, etc. The traditional labeling is: Andre=1, Beverley=2, Carol=3, Diane=4, Ed=5, Fernando=6, Garth=7, Heather=8, Ike=9, Jane=10.

6.3.15 networkx.generators.small.moebius_kantor_graph

networkx.generators.small.**moebius_kantor_graph**(*create_using=None*)
Return the Moebius-Kantor graph.

6.3.16 networkx.generators.small.octahedral_graph

networkx.generators.small.**octahedral_graph**(*create_using=None*)
Return the Platonic Octahedral graph.

6.3.17 networkx.generators.small.pappus_graph

networkx.generators.small.**pappus_graph**()
Return the Pappus graph.

6.3.18 networkx.generators.small.petersen_graph

networkx.generators.small.**petersen_graph**(*create_using=None*)
Return the Petersen graph.

6.3.19 networkx.generators.small.sedgewick_maze_graph

networkx.generators.small.**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. Nodes are numbered 0,..,7

6.3.20 networkx.generators.small.tetrahedral_graph

networkx.generators.small.**tetrahedral_graph**(*create_using=None*)
Return the 3-regular Platonic Tetrahedral graph.

6.3.21 networkx.generators.small.truncated_cube_graph

networkx.generators.small.**truncated_cube_graph**(*create_using=None*)
Return the skeleton of the truncated cube.

6.3.22 networkx.generators.small.truncated_tetrahedron_graph

networkx.generators.small.**truncated_tetrahedron_graph**(*create_using=None*)
Return the skeleton of the truncated Platonic tetrahedron.

6.3.23 networkx.generators.small.tutte_graph

networkx.generators.small.**tutte_graph**(*create_using=None*)
Return the Tutte graph.

6.4 Random Graphs

Generators for random graphs.

fast_gnp_random_graph(n, p[, seed, directed])	Return a random graph G_{n,p} (Erdős-Rényi graph, binomial graph).
gnp_random_graph(n, p[, seed, directed])	Return a random graph G_{n,p} (Erdős-Rényi graph, binomial graph).
dense_gnm_random_graph(n, m[, seed])	Return the random graph G_{n,m}.
gnm_random_graph(n, m[, seed, directed])	Return the random graph G_{n,m}.
erdos_renyi_graph(n, p[, seed, directed])	Return a random graph G_{n,p} (Erdős-Rényi graph, binomial graph).
binomial_graph(n, p[, seed, directed])	Return a random graph G_{n,p} (Erdős-Rényi graph, binomial graph).
newman_watts_strogatz_graph(n, k, p[, ...])	Return a Newman-Watts-Strogatz small world graph.
watts_strogatz_graph(n, k, p[, ...])	Return a Watts-Strogatz small-world graph.
connected_watts_strogatz_graph(n, k, p[, ...])	Return a connected Watts-Strogatz small-world graph.
random_regular_graph(d, n[, create_using, seed])	Return a random regular graph of n nodes each with degree d.
barabasi_albert_graph(n, m[, create_using, seed])	Return random graph using Barabási-Albert preferential attachment model.
powerlaw_cluster_graph(n, m, p[, ...])	Holme and Kim algorithm for growing graphs with powerlaw
random_lobster(n, p1, p2[, create_using, seed])	Return a random lobster.
random_shell_graph(constructor[, ...])	Return a random shell graph for the constructor given.
random_powerlaw_tree(n[, gamma, ...])	Return a tree with a powerlaw degree distribution.
random_powerlaw_tree_sequence(n[, gamma, ...])	Return a degree sequence for a tree with a powerlaw distribution.

6.4.1 networkx.generators.random_graphs.fast_gnp_random_graph

networkx.generators.random_graphs.**fast_gnp_random_graph**(*n*, *p*, *seed=None*, *directed=False*)
Return a random graph G_{n,p} (Erdős-Rényi graph, binomial graph).

Parameters **n** : int

The number of nodes.

p : float

Probability for edge creation.

seed : int, optional

Seed for random number generator (default=None).

directed : bool, optional (default=False)

If True return 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 is O(n+m) where m is the expected number of edges m=p*n*(n-1)/2.

It should be faster than gnp_random_graph when p is small and the expected number of edges is small (sparse graph).

References

[R178]

6.4.2 networkx.generators.random_graphs.gnp_random_graph

networkx.generators.random_graphs.**gnp_random_graph**(*n*, *p*, *seed=None*, *directed=False*)

Return a random graph G_{n,p} (Erdős-Rényi graph, binomial graph).

Chooses each of the possible edges with probability p.

This is also called binomial_graph and erdos_renyi_graph.

Parameters **n** : int

The number of nodes.

p : float

Probability for edge creation.

seed : int, optional

Seed for random number generator (default=None).

directed : bool, optional (default=False)

If True return a directed graph

See Also:

fast_gnp_random_graph

Notes

This is an O(n^2) algorithm. For sparse graphs (small p) see fast_gnp_random_graph for a faster algorithm.

References

[R179], [R180]

6.4.3 networkx.generators.random_graphs.dense_gnm_random_graph

networkx.generators.random_graphs.**dense_gnm_random_graph**(*n*, *m*, *seed=None*)

Return the random graph G_{n,m}.

Gives a graph picked randomly out of 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 : int, optional

Seed for random number generator (default=None).

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 [R175].

References

[R175]

6.4.4 networkx.generators.random_graphs.gnm_random_graph

networkx.generators.random_graphs.**gnm_random_graph**(*n*, *m*, *seed=None*, *directed=False*)

Return the random graph G_{n,m}.

Produces a graph picked randomly out of the set of all graphs with n nodes and m edges.

Parameters **n** : int

The number of nodes.

m : int

The number of edges.

seed : int, optional

Seed for random number generator (default=None).

directed : bool, optional (default=False)

If True return a directed graph

6.4.5 networkx.generators.random_graphs.erdos_renyi_graph

networkx.generators.random_graphs.**erdos_renyi_graph**(*n*, *p*, *seed=None*, *directed=False*)

Return a random graph G_{n,p} (Erdős-Rényi graph, binomial graph).

Chooses each of the possible edges with probability p.

This is also called binomial_graph and erdos_renyi_graph.

Parameters **n** : int

The number of nodes.

p : float

Probability for edge creation.

seed : int, optional

Seed for random number generator (default=None).

directed : bool, optional (default=False)

If True return a directed graph

See Also:

fast_gnp_random_graph

Notes

This is an O(n^2) algorithm. For sparse graphs (small p) see fast_gnp_random_graph for a faster algorithm.

References

[R176], [R177]

6.4.6 networkx.generators.random_graphs.binomial_graph

networkx.generators.random_graphs.**binomial_graph**(*n*, *p*, *seed=None*, *directed=False*)

Return a random graph G_{n,p} (Erdős-Rényi graph, binomial graph).

Chooses each of the possible edges with probability p.

This is also called binomial_graph and erdos_renyi_graph.

Parameters **n** : int

The number of nodes.

p : float

Probability for edge creation.

seed : int, optional

Seed for random number generator (default=None).

directed : bool, optional (default=False)

If True return a directed graph

See Also:

fast_gnp_random_graph

Notes

This is an O(n^2) algorithm. For sparse graphs (small p) see fast_gnp_random_graph for a faster algorithm.

References

[R173], [R174]

6.4.7 networkx.generators.random_graphs.newman_watts_strogatz_graph

networkx.generators.random_graphs.**newman_watts_strogatz_graph**(*n*, *k*, *p*, *create_using=None*, *seed=None*)

Return a Newman-Watts-Strogatz small world graph.

Parameters **n** : int

The number of nodes

k : int

Each node is connected to k nearest neighbors in ring topology

p : float

The probability of adding a new edge for each edge

create_using : graph, optional (default Graph)

The graph instance used to build the graph.

seed : int, optional

seed for random number generator (default=None)

See Also:

watts_strogatz_graph

Notes

First create a ring over n nodes. Then each node in the ring is connected with its k nearest neighbors (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

[R181]

6.4.8 networkx.generators.random_graphs.watts_strogatz_graph

networkx.generators.random_graphs.**watts_strogatz_graph**(*n*, *k*, *p*, *create_using=None*, *seed=None*)

Return a Watts-Strogatz small-world graph.

Parameters **n** : int

The number of nodes

k : int

Each node is connected to k nearest neighbors in ring topology

p : float

The probability of rewiring each edge

create_using : graph, optional (default Graph)

The graph instance used to build the graph.

seed : int, optional

Seed for random number generator (default=None)

See Also:

newman_watts_strogatz_graph, connected_watts_strogatz_graph

Notes

First create a ring over n nodes. Then each node in the ring is connected with its k nearest neighbors (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

[R185]

6.4.9 networkx.generators.random_graphs.connected_watts_strogatz_graph

networkx.generators.random_graphs.**connected_watts_strogatz_graph**(*n*, *k*, *p*, *tries=100*, *create_using=None*, *seed=None*)

Return a connected Watts-Strogatz small-world graph.

Attempt to generate a connected realization 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 connected to k nearest neighbors in ring topology

p : float

The probability of rewiring each edge

tries : int

Number of attempts to generate a connected graph.

create_using : graph, optional (default Graph)

The graph instance used to build the graph.

seed : int, optional

The seed for random number generator.

See Also:

newman_watts_strogatz_graph, watts_strogatz_graph

6.4.10 networkx.generators.random_graphs.random_regular_graph

networkx.generators.random_graphs.**random_regular_graph**(*d*, *n*, *create_using=None*, *seed=None*)

Return a random regular graph of n nodes each with degree d.

The resulting graph G has no self-loops or parallel edges.

Parameters **d** : int

Degree

n : integer

Number of nodes. The value of n*d must be even.

create_using : graph, optional (default Graph)

The graph instance used to build the graph.

seed : hashable object

The seed for random number generator.

Notes

The nodes are numbered form 0 to n-1.

Kim and Vu's paper [R184] shows that this algorithm samples in an asymptotically uniform way from the space of random graphs when d = O(n**(1/3-epsilon)).

References

[R183], [R184]

6.4.11 networkx.generators.random_graphs.barabasi_albert_graph

networkx.generators.random_graphs.**barabasi_albert_graph**(*n*, *m*, *create_using=None*, *seed=None*)

Return random graph using Barabási-Albert preferential attachment model.

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

create_using : graph, optional (default Graph)

The graph instance used to build the graph.

seed : int, optional

Seed for random number generator (default=None).

Returns **G** : Graph

Notes

The initialization is a graph with with m nodes and no edges.

References

[R172]

6.4.12 networkx.generators.random_graphs.powerlaw_cluster_graph

networkx.generators.random_graphs.**powerlaw_cluster_graph**(*n*, *m*, *p*, *create_using=None*, *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

create_using : graph, optional (default Graph)

The graph instance used to build the graph.

seed : int, optional

Seed for random number generator (default=None).

Notes

The average clustering has a hard time getting above a certain cutoff that depends on m. This cutoff is often quite low. Note that the transitivity (fraction of triangles to possible triangles) seems to go down with network size.

It is essentially the Barabási-Albert (B-A) 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 B-A 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 B-A model.

References

[R182]

6.4.13 networkx.generators.random_graphs.random_lobster

networkx.generators.random_graphs.**random_lobster**(*n*, *p1*, *p2*, *create_using=None*, *seed=None*)

Return a random lobster.

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 (p2=0).

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

create_using : graph, optional (default Graph)

The graph instance used to build the graph.

seed : int, optional

Seed for random number generator (default=None).

6.4.14 networkx.generators.random_graphs.random_shell_graph

networkx.generators.random_graphs.**random_shell_graph**(*constructor*, *create_using=None*, *seed=None*)

Return a random shell graph for the constructor given.

Parameters **constructor: a list of three-tuples** :

(n,m,d) for each shell starting at the center shell.

n : int

The number of nodes in the shell

m : int

The number or edges in the shell

d : float

The ratio of inter-shell (next) edges to intra-shell edges. d=0 means no intra shell edges, d=1 for the last shell

create_using : graph, optional (default Graph)

The graph instance used to build the graph.

seed : int, optional

Seed for random number generator (default=None).

Examples

```
>>> constructor=[(10,20,0.8),(20,40,0.8)]
>>> G=nx.random_shell_graph(constructor)
```

6.4.15 networkx.generators.random_graphs.random_powerlaw_tree

networkx.generators.random_graphs.**random_powerlaw_tree**(*n*, *gamma=3*, *create_using=None*, *seed=None*, *tries=100*)

Return a tree with a powerlaw degree distribution.

Parameters **n** : int,

The number of nodes

gamma : float

Exponent of the power-law

create_using : graph, optional (default Graph)

The graph instance used to build the graph.

seed : int, optional

Seed for random number generator (default=None).

tries : int

Number of attempts to adjust sequence to make a tree

Notes

A trial powerlaw degree sequence is chosen and then elements are swapped with new elements from a powerlaw distribution until the sequence makes a tree (#edges=#nodes-1).

6.4.16 networkx.generators.random_graphs.random_powerlaw_tree_sequence

networkx.generators.random_graphs.**random_powerlaw_tree_sequence**(*n*, *gamma=3*, *seed=None*, *tries=100*)

Return a degree sequence for a tree with a powerlaw distribution.

Parameters **n** : int,

The number of nodes

gamma : float

Exponent of the power-law

seed : int, optional

Seed for random number generator (default=None).

tries : int

Number of attempts to adjust sequence to make a tree

Notes

A trial powerlaw degree sequence is chosen and then elements are swapped with new elements from a powerlaw distribution until the sequence makes a tree (#edges=#nodes-1).

6.5 Degree Sequence

Generate graphs with a given degree sequence or expected degree sequence.

configuration_model(deg_sequence[, ...])	Return a random graph with the given degree sequence.
directed_configuration_model(...[, ...])	Return a directed_random graph with the given degree sequences.
expected_degree_graph(w[, seed, selfloops])	Return a random graph with given expected degrees.
havel_hakimi_graph(deg_sequence[, create_using])	Return a simple graph with given degree sequence and constructed
degree_sequence_tree(deg_sequence[, ...])	Make a tree for the given degree sequence.
is_valid_degree_sequence_havel_hakimi(...)	Returns True if deg_sequence is a valid degree sequence.
is_valid_degree_sequence_erdos_gallai(...)	Returns True if deg_sequence is a valid degree sequence.
create_degree_sequence(n, **kwds[, ...])	Attempt to create a valid degree sequence of length n using specified function sfunction(n,**kwds).
double_edge_swap(G[, nswap, max_tries])	Swap two edges in the graph while keeping the node degrees fixed.
connected_double_edge_swap(G[, nswap])	Attempt nswap double-edge swaps on the graph G.
li_smax_graph(degree_seq[, create_using])	Generates a graph based with a given degree sequence and maximizing the s-metric.
random_clustered_graph(joint_degree_sequence)	Generate a random graph with the given joint degree and triangle degree sequence.

6.5.1 networkx.generators.degree_seq.configuration_model

networkx.generators.degree_seq.**configuration_model**(*deg_sequence*, *create_using=None*, *seed=None*)

Return 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 integers

Each list entry corresponds to the degree of a node.

create_using : graph, optional (default MultiGraph)

Return graph of this type. The instance will be cleared.

seed : hashable object, optional

Seed for random number generator.

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_valid_degree_sequence

Notes

As described by Newman [R151].

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. This "finite-size effect" decreases as the size of the graph increases.

References

[R151]

Examples

```
>>> from networkx.utils import powerlaw_sequence
>>> z=nx.create_degree_sequence(100,powerlaw_sequence)
>>> G=nx.configuration_model(z)
```

To remove parallel edges:

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

To remove self loops:

```
>>> G.remove_edges_from(G.selfloop_edges())
```

6.5.2 networkx.generators.degree_seq.directed_configuration_model

networkx.generators.degree_seq.**directed_configuration_model**(*in_degree_sequence*, *out_degree_sequence*, *create_using=None*, *seed=None*)

Return 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 integers

Each list entry corresponds to the in-degree of a node.

out_degree_sequence : list of integers

Each list entry corresponds to the out-degree of a node.

create_using : graph, optional (default MultiDiGraph)

Return graph of this type. The instance will be cleared.

seed : hashable object, optional

Seed for random number generator.

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 [R153].

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

[R153]

Examples

```
>>> D=nx.DiGraph([(0,1),(1,2),(2,3)]) # directed path graph
>>> din=list(D.in_degree().values())
>>> dout=list(D.out_degree().values())
>>> din.append(1)
>>> dout[0]=2
>>> D=nx.directed_configuration_model(din,dout)
```

To remove parallel edges:

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

To remove self loops:

```
>>> D.remove_edges_from(D.selfloop_edges())
```

6.5.3 networkx.generators.degree_seq.expected_degree_graph

networkx.generators.degree_seq.**expected_degree_graph**(*w*, *seed=None*, *self-loops=True*)

Return 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 : hashable object, optional

The seed for the random number generator.

Returns **Graph** :

Notes

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 [R154] 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

[R154], [R155]

Examples

```
>>> z=[10 for i in range(100)]
>>> G=nx.expected_degree_graph(z)
```

6.5.4 networkx.generators.degree_seq.havel_hakimi_graph

networkx.generators.degree_seq.**havel_hakimi_graph**(*deg_sequence*, *create_using=None*)

Return a simple graph with given degree sequence and 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 : graph, optional (default Graph)

Return graph of this type. The instance will be cleared. Multigraphs and 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.

See Theorem 1.4 in [R156]. This algorithm is also used in the function is_valid_degree_sequence.

References

[R156]

6.5.5 networkx.generators.degree_seq.degree_sequence_tree

networkx.generators.degree_seq.**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

6.5.6 networkx.generators.degree_seq.is_valid_degree_sequence_havel_hakimi

networkx.generators.degree_seq.**is_valid_degree_sequence_havel_hakimi**(*deg_sequence*)

Returns True if deg_sequence is a valid degree sequence.

A degree sequence is valid if some graph can realize it. Validation proceeds via the Havel-Hakimi algorithm.

Worst-case run time is: O(n**(log n))

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 a valid degree sequence and False if not.

References

[havel1955], [hakimi1962], [CL1996]

6.5.7 networkx.generators.degree_seq.is_valid_degree_sequence_erdos_gallai

networkx.generators.degree_seq.**is_valid_degree_sequence_erdos_gallai**(*deg_sequence*)

Returns True if deg_sequence is a valid degree sequence.

A degree sequence is valid if some graph can realize it. Validation proceeds via the Erdős-Gallai algorithm.

Worst-case run time is: O(n**2)

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 a valid degree sequence and False if not.

References

[EG1960], [choudum1986]

6.5.8 networkx.generators.degree_seq.create_degree_sequence

networkx.generators.degree_seq.**create_degree_sequence**(*n*, *sfunction=None*, *max_tries=50*, ***kwds*)

Attempt to create a valid degree sequence of length n using specified function sfunction(n,**kwds).

Parameters **n** : int

Length of degree sequence = number of nodes

sfunction: function :

Function which returns a list of n real or integer values. Called as "sfunction(n,**kwds)".

max_tries: int :

Max number of attempts at creating valid degree sequence.

Notes

Repeatedly create a degree sequence by calling sfunction(n,**kwds) until achieving a valid degree sequence. If unsuccessful after max_tries attempts, raise an exception.

For examples of sfunctions that return sequences of random numbers, see networkx.Utils.

Examples

```
>>> from networkx.utils import uniform_sequence
>>> seq=nx.create_degree_sequence(10,uniform_sequence)
```

6.5.9 networkx.generators.degree_seq.double_edge_swap

networkx.generators.degree_seq.**double_edge_swap**(*G*, *nswap=1*, *max_tries=100*)

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

A NetworkX (undirected) Graph.

nswap : integer (optional)

Number of double-edge swaps to perform

max_tries : integer (optional)

Maximum number of attempts to swap nswap edges.

Returns **G** : graph

The graph after nswap double edge swaps.

Notes

Does not enforce any connectivity constraints.

The graph G is modified in place.

6.5.10 networkx.generators.degree_seq.connected_double_edge_swap

networkx.generators.degree_seq.**connected_double_edge_swap**(*G*, *nswap=1*)

Attempt nswap double-edge swaps on the graph G.

Returns the count of successful swaps. Enforces connectivity. The graph G is modified in place.

Notes

A double-edge swap removes two randomly choseen 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 so the actual count of swapped edges is always <= nswap

The initial graph G must be connected and the resulting graph is connected.

References

[R152]

6.5.11 networkx.generators.degree_seq.li_smax_graph

networkx.generators.degree_seq.**li_smax_graph**(*degree_seq*, *create_using=None*)

Generates a graph based with a given degree sequence and maximizing the s-metric. Experimental implementation.

Maximum s-metrix means that high degree nodes are connected to high degree nodes.

- $degree_seq$**: degree sequence, a list of integers with each entry** corresponding to the degree of a node. A non-graphical degree sequence raises an Exception.

Reference:

```
@unpublished{li-2005,
 author = {Lun Li and David Alderson and Reiko Tanaka
          and John C. Doyle and Walter Willinger},
 title = {Towards a Theory of Scale-Free Graphs:
         Definition, Properties, and  Implications (Extended Version)},
 url = {http://arxiv.org/abs/cond-mat/0501169},
 year = {2005}
}
```

The algorithm:

```
STEP 0 - Initialization
A = {0}
B = {1, 2, 3, ..., n}
O = {(i; j), ..., (k, l),...} where i < j, i <= k < l and
        d_i * d_j >= d_k *d_l
wA = d_1
dB = sum(degrees)

STEP 1 - Link selection
```

```
(a) If |O| = 0 TERMINATE. Return graph A.
(b) Select element(s) (i, j) in O having the largest d_i * d_j , if for
        any i or j either w_i = 0 or w_j = 0 delete (i, j) from O
(c) If there are no elements selected go to (a).
(d) Select the link (i, j) having the largest value w_i (where for each
        (i, j) w_i is the smaller of w_i and w_j ), and proceed to STEP 2.

STEP 2 - Link addition
Type 1: i in A and j in B.
        Add j to the graph A and remove it from the set B add a link
        (i, j) to the graph A. Update variables:
        wA = wA + d_j -2 and dB = dB - d_j
        Decrement w_i and w_j with one. Delete (i, j) from O
Type 2: i and j in A.
    Check Tree Condition: If dB = 2 * |B| - wA.
        Delete (i, j) from O, continue to STEP 3
    Check Disconnected Cluster Condition: If wA = 2.
        Delete (i, j) from O, continue to STEP 3
    Add the link (i, j) to the graph A
    Decrement w_i and w_j with one, and wA = wA -2
STEP 3
    Go to STEP 1
```

The article states that the algorithm will result in a maximal s-metric. This implementation can not guarantee such maximality. I may have misunderstood the algorithm, but I can not see how it can be anything but a heuristic. Please contact me at sundsdal@gmail.com if you can provide python code that can guarantee maximality. Several optimizations are included in this code and it may be hard to read. Commented code to come.

A POSSIBLE ALTERNATIVE:

For an 'unconstrained' graph, that is one they describe as having the sum of the degree sequence be even(ie all undirected graphs) they present a simpler algorithm. It is as follows

> "For each vertex i: if di is even then attach di/2 self-loops; if di is odd, then attach (di-1)/2 self-loops, leaving one available "stub". Second for all remaining vertices with "stubs" connect them in pairs according to decreasing values of di."[1]

Since this only works for undirected graphs anyway, perhaps this is the better method? Note this also returns a graph with a larger s_metric than the other method, and it seems to have the same degree sequence, though I haven't tested it extensively.

6.5.12 networkx.generators.degree_seq.random_clustered_graph

networkx.generators.degree_seq.**random_clustered_graph**(*joint_degree_sequence*, *create_using=None*, *seed=None*)

Generate a random graph with the given joint degree and triangle degree sequence.

This uses a configuration model-like approach to generate a random pseudograph (graph with parallel edges and self loops) by randomly assigning edges to match the given indepdenent edge and triangle degree sequence.

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 : graph, optional (default MultiGraph)

Return graph of this type. The instance will be cleared.

seed : hashable object, optional

The seed for the random number generator.

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 [R157] (see also Newman [R158] 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

[R157], [R158]

Examples

```
>>> deg_tri=[[1,0],[1,0],[1,0],[2,0],[1,0],[2,1],[0,1],[0,1]]
>>> G = nx.random_clustered_graph(deg_tri)
```

To remove parallel edges:

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

To remove self loops:

```
>>> G.remove_edges_from(G.selfloop_edges())
```

6.6 Directed

Generators for some directed graphs.

gn_graph: growing network gnc_graph: growing network with copying gnr_graph: growing network with redirection scale_free_graph: scale free directed graph

gn_graph(n[, kernel, create_using, seed])	Return the GN digraph with n nodes.
gnr_graph(n, p[, create_using, seed])	Return the GNR digraph with n nodes and redirection probability p.
gnc_graph(n[, create_using, seed])	Return the GNC digraph with n nodes.
scale_free_graph(n[, alpha, beta, gamma, ...])	Return a scale free directed graph.

6.6.1 networkx.generators.directed.gn_graph

networkx.generators.directed.**gn_graph**(*n*, *kernel=None*, *create_using=None*, *seed=None*)

Return the GN digraph with n nodes.

The GN (growing network) 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 degree.

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 : graph, optional (default DiGraph)

Return graph of this type. The instance will be cleared.

seed : hashable object, optional

The seed for the random number generator.

References

[R159]

Examples

```
>>> D=nx.gn_graph(10)     # the GN graph
>>> G=D.to_undirected()   # the undirected version
```

To specify an attachment kernel use the kernel keyword

```
>>> D=nx.gn_graph(10,kernel=lambda x:x**1.5) # A_k=k^1.5
```

6.6.2 networkx.generators.directed.gnr_graph

networkx.generators.directed.**gnr_graph**(*n*, *p*, *create_using=None*, *seed=None*)

Return the GNR digraph with n nodes and redirection probability p.

The GNR (growing network with redirection) 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 : graph, optional (default DiGraph)

Return graph of this type. The instance will be cleared.

seed : hashable object, optional

The seed for the random number generator.

References

[R161]

Examples

```
>>> D=nx.gnr_graph(10,0.5)  # the GNR graph
>>> G=D.to_undirected()  # the undirected version
```

6.6.3 networkx.generators.directed.gnc_graph

networkx.generators.directed.**gnc_graph**(*n*, *create_using=None*, *seed=None*)

Return the GNC digraph with n nodes.

The GNC (growing network with copying) graph is built by adding nodes one at a time with a links 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 : graph, optional (default DiGraph)

Return graph of this type. The instance will be cleared.

seed : hashable object, optional

The seed for the random number generator.

References

[R160]

6.6.4 networkx.generators.directed.scale_free_graph

networkx.generators.directed.**scale_free_graph**(*n*, *alpha=0.40999999999999998*, *beta=0.54000000000000004*, *gamma=0.050000000000000003*, *delta_in=0.20000000000000001*, *delta_out=0*, *create_using=None*, *seed=None*)

Return 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 conecgted to an existing node chosen randomly according to the out-degree distribution.

delta_in : float

Bias for choosing ndoes from in-degree distribution.

delta_out : float

Bias for choosing ndoes from out-degree distribution.

create_using : graph, optional (default MultiDiGraph)

Use this graph instance to start the process (default=3-cycle).

seed : integer, optional

Seed for random number generator

Notes

The sum of alpha, beta, and gamma must be 1.

References

[R162]

Examples

```
>>> G=nx.scale_free_graph(100)
```

6.7 Geometric

Generators for geometric graphs.

random_geometric_graph(n, radius[, dim, pos])	Return the random geometric graph in the unit cube.
geographical_threshold_graph(n, theta[, ...])	Return a geographical threshold graph.
waxman_graph(n, 0, 1[, alpha, beta, L, domain])	Return a Waxman random graph.
navigable_small_world_graph(n[, p, q, r, ...])	Return a navigable small-world graph.

6.7.1 networkx.generators.geometric.random_geometric_graph

networkx.generators.geometric.**random_geometric_graph**(*n*, *radius*, *dim=2*, *pos=None*)

Return the random geometric graph in the unit cube.

The random geometric graph model places n nodes uniformly at random in the unit cube Two nodes u, v are connected with an edge if $d(u, v) <= r$ where d is the Euclidean distance and r is a radius threshold.

Parameters **n** : int

Number 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.

Returns **Graph** :

Notes

This uses an n^2 algorithm to build the graph. A faster algorithm is possible using k-d trees.

The pos keyword can be used to specify node positions so you can create an arbitrary distribution and domain for positions. If you need a distance function other than Euclidean you'll have to hack the algorithm.

E.g to use a 2d Gaussian distribution of node positions with mean (0,0) and std. dev. 2

```
>>> import random
>>> n=20
>>> p=dict((i,(random.gauss(0,2),random.gauss(0,2))) for i in range(n))
>>> G = nx.random_geometric_graph(n,0.2,pos=p)
```

References

[R166]

Examples

```
>>> G = nx.random_geometric_graph(20,0.1)
```

6.7.2 networkx.generators.geometric.geographical_threshold_graph

networkx.generators.geometric.**geographical_threshold_graph**(*n*, *theta*, *alpha=2*, *dim=2*, *pos=None*, *weight=None*)

Return 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, v are connected with an edge if

$$w_u + w_v \geq \theta r^\alpha$$

where r is the Euclidean distance between u and v, and θ, α are parameters.

Parameters **n** : int

Number of nodes

theta: float :

Threshold value

alpha: float, optional :

Exponent of distance function

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.

Returns **Graph** :

Notes

If weights are not specified they are assigned to nodes by drawing randomly from an the exponential distribution with rate parameter $\lambda = 1$. To specify a weights from a different distribution assign them to a dictionary and pass it as the weight= keyword

```
>>> import random
>>> n = 20
>>> w=dict((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

[R163], [R164]

Examples

```
>>> G = nx.geographical_threshold_graph(20,50)
```

6.7.3 networkx.generators.geometric.waxman_graph

networkx.generators.geometric.**waxman_graph**(*n*, *alpha=0.40000000000000002*, *beta=0.10000000000000001*, *L=None*, *domain=(0, 0, 1, 1)*)

Return a Waxman random graph.

The Waxman random graph models place n nodes uniformly at random in a rectangular domain. Two nodes u,v are connected with an edge with probability

$$p = \alpha * exp(d/(\beta * L)).$$

This function implements both Waxman models.

Waxman-1: L not specified The distance d is the Euclidean distance between the nodes u and v. L is the maximum distance between all nodes in the graph.

Waxman-2: L specified The distance d is chosen randomly in $[0, L]$.

Parameters **n** : int

Number of nodes

alpha: float :

Model parameter

beta: float :

Model parameter

L : float, optional

Maximum distance between nodes. If not specified the actual distance is calculated.

domain : tuple of numbers, optional

Domain size (xmin, ymin, xmax, ymax)

Returns **G: Graph** :

References

[R167]

6.7.4 networkx.generators.geometric.navigable_small_world_graph

networkx.generators.geometric.**navigable_small_world_graph**(*n*, *p=1*, *q=1*, *r=2*, *dim=2*, *seed=None*)

Return a navigable small-world graph.

A navigable small-world graph is a directed grid with additional long-range connections that are chosen randomly. From [R165]:

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 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, the node u has a directed edge to every other node within lattice distance p (local contacts) .

For universal constants $q \geq 0$ and $r \geq 0$ construct directed edges from u to q other nodes (long-range contacts) using independent random trials; the i'th directed edge from u has endpoint v with probability proportional to $d(u,v)^{-r}$.

Parameters **n** : int

The number of nodes.

p : int

The diameter of short range connections. Each node is connected to every other node within lattice distance p.

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 : int, optional

Seed for random number generator (default=None).

References

[R165]

6.8 Hybrid

Hybrid

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 G is kl connected.

6.8.1 networkx.generators.hybrid.kl_connected_subgraph

networkx.generators.hybrid.**kl_connected_subgraph**(*G*, *k*, *l*, *low_memory=False*, *same_as_graph=False*)

Returns the maximum locally (k,l) connected subgraph of G.

(k,l)-connected subgraphs are presented by Fan Chung and Li in "The Small World Phenomenon in hybrid power law graphs" to appear in "Complex Networks" (Ed. E. Ben-Naim) Lecture Notes in Physics, Springer (2004)

low_memory=True then use a slightly slower, but lower memory version same_as_graph=True then return a tuple with subgraph and pflag for if G is kl-connected

6.8.2 networkx.generators.hybrid.is_kl_connected

networkx.generators.hybrid.**is_kl_connected**(*G*, *k*, *l*, *low_memory=False*)

Returns True if G is kl connected.

6.9 Bipartite

Generators and functions for bipartite graphs.

bipartite_configuration_model(aseq, bseq[, ...])	Return a random bipartite graph from two given degree sequences.
bipartite_havel_hakimi_graph(aseq, bseq[, ...])	Return a bipartite graph from two given degree sequences using a
bipartite_reverse_havel_hakimi_graph(aseq, bseq)	Return a bipartite graph from two given degree sequences using a
bipartite_alternating_havel_hakimi_graph(aseq, bseq)	Return a bipartite graph from two given degree sequences using
bipartite_preferential_attachment_graph(aseq, p)	Create a bipartite graph with a preferential attachment model from a given single degree sequence.
bipartite_random_regular_graph(d, n[, ...])	Experimental: Generate a random regular bipartite graph.
bipartite_random_graph(n, m, p[, seed, directed])	Return a bipartite random graph.

6.9.1 networkx.generators.bipartite.bipartite_configuration_model

networkx.generators.bipartite.**bipartite_configuration_model**(*aseq*, *bseq*, *create_using=None*, *seed=None*)

Return a random bipartite graph from two given degree sequences.

Parameters **aseq** : list or iterator

Degree sequence for node set A.

bseq : list or iterator

Degree sequence for node set B.

create_using : NetworkX graph instance, optional

Return graph of this type.

seed : integer, optional

Seed for random number generator.

Nodes from the set A are connected to nodes in the 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.

6.9.2 networkx.generators.bipartite.bipartite_havel_hakimi_graph

networkx.generators.bipartite.**bipartite_havel_hakimi_graph**(*aseq*, *bseq*, *create_using=None*)

Return a bipartite graph from two given degree sequences using a Havel-Hakimi style construction.

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 or iterator

Degree sequence for node set A.

bseq : list or iterator

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.

6.9.3 networkx.generators.bipartite.bipartite_reverse_havel_hakimi_graph

networkx.generators.bipartite.**bipartite_reverse_havel_hakimi_graph**(*aseq*, *bseq*, *create_using=None*)

Return a bipartite graph from two given degree sequences using a Havel-Hakimi style construction.

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 or iterator

Degree sequence for node set A.

bseq : list or iterator

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.

6.9.4 networkx.generators.bipartite.bipartite_alternating_havel_hakimi_graph

networkx.generators.bipartite.**bipartite_alternating_havel_hakimi_graph**(*aseq*, *bseq*, *create_using=None*)

Return a bipartite graph from two given degree sequences using an alternating Havel-Hakimi style construction.

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 or iterator

Degree sequence for node set A.

bseq : list or iterator

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.

6.9.5 networkx.generators.bipartite.bipartite_preferential_attachment_graph

networkx.generators.bipartite.**bipartite_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.

Parameters **aseq** : list or iterator

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, optional

Seed for random number generator.

References

[R149]

6.9.6 networkx.generators.bipartite.bipartite_random_regular_graph

networkx.generators.bipartite.**bipartite_random_regular_graph**(*d*, *n*, *create_using=None*, *seed=None*)

Experimental: Generate a random regular bipartite graph.

Parameters **d** : integer

Degree of graph.

n : integer

Number of nodes in graph.

create_using : NetworkX graph instance, optional

Return graph of this type.

seed : integer, optional

Seed for random number generator.

Notes

This is an untested, unproved algorithm.

Nodes are numbered 0...n-1.

Restrictions on n and d:

- n must be even
- n>=2*d

Algorithm inspired by random_regular_graph()

6.9.7 networkx.generators.bipartite.bipartite_random_graph

networkx.generators.bipartite.**bipartite_random_graph**(*n*, *m*, *p*, *seed=None*, *directed=False*)

Return a bipartite random graph.

This is a bipartite version of the binomial (Erdős-Rényi) graph.

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 : int, optional

Seed for random number generator (default=None).

directed : bool, optional (default=False)

If True return a directed graph

See Also:

gnp_random_graph, bipartite_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.

References

[R150]

6.10 Line Graph

Line graphs.

line_graph(G)	Return the line graph of the graph or digraph G.

6.10.1 networkx.generators.line.line_graph

networkx.generators.line.**line_graph**(*G*)

Return 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 between those nodes if the two edges in G share a common node.

For DiGraphs an edge an edge represents a directed path of length 2.

The original node labels are kept as two-tuple node labels in the line graph.

Parameters **G** : graph

A NetworkX Graph or DiGraph

Notes

Not implemented for MultiGraph or MultiDiGraph classes.

Graph, node, and edge data are not propagated to the new graph.

Examples

```
>>> G=nx.star_graph(3)
>>> L=nx.line_graph(G)
>>> print(sorted(L.edges())) # makes a clique, K3
[((0, 1), (0, 2)), ((0, 1), (0, 3)), ((0, 3), (0, 2))]
```

6.11 Ego Graph

Ego graph.

ego_graph(G, n[, radius, center, ...])	Returns induced subgraph of neighbors centered at node n within a given radius.

6.11.1 networkx.generators.ego.ego_graph

networkx.generators.ego.**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.

6.12 Stochastic

Stocastic graph.

stochastic_graph(G[, copy])	Return a right-stochastic representation of G.

6.12.1 networkx.generators.stochastic.stochastic_graph

networkx.generators.stochastic.**stochastic_graph**(*G*, *copy=True*)

Return a right-stochastic representation of G.

A right-stochastic graph is a weighted graph in which all of the node (out) neighbors edge weights sum to 1.

Parameters **G** : graph

A NetworkX graph, must have valid edge weights

copy : boolean, optional

If True make a copy of the graph, otherwise modify original graph

6.13 Intersection

Generators for random intersection graphs.

uniform_random_intersection_graph(n, m, p[, ...])	Return a uniform random intersection graph.
k_random_intersection_graph(n, m, k)	Return a intersection graph with randomly chosen attribute sets for each node that are of equal size (k).
general_random_intersection_graph(n, m, p)	Return a random intersection graph with independent probabilities for connections between node and attribute sets.

6.13.1 networkx.generators.intersection.uniform_random_intersection_graph

networkx.generators.intersection.**uniform_random_intersection_graph**(*n*, *m*, *p*, *seed=None*)

Return 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 : int, optional

Seed for random number generator (default=None).

See Also:

gnp_random_graph

References

[R170], [R171]

6.13.2 networkx.generators.intersection.k_random_intersection_graph

networkx.generators.intersection.**k_random_intersection_graph**(*n*, *m*, *k*)

Return 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 : int, optional

Seed for random number generator (default=None).

See Also:

gnp_random_graph, uniform_random_intersection_graph

References

[R169]

6.13.3 networkx.generators.intersection.general_random_intersection_graph

networkx.generators.intersection.**general_random_intersection_graph**(*n*, *m*, *p*)

Return 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 : int, optional

Seed for random number generator (default=None).

See Also:

gnp_random_graph, uniform_random_intersection_graph

References

[R168]

6.14 Social Networks

Famous social networks.

karate_club_graph()	Return Zachary's Karate club graph.
davis_southern_women_graph()	Return Davis Southern women social network.
florentine_families_graph()	Return Florentine families graph.

6.14.1 networkx.generators.social.karate_club_graph

networkx.generators.social.**karate_club_graph**()
Return Zachary's Karate club graph.

References

[R188], [R189]

6.14.2 networkx.generators.social.davis_southern_women_graph

networkx.generators.social.**davis_southern_women_graph**()
Return Davis Southern women social network.

This is a bipartite graph.

References

[R186]

6.14.3 networkx.generators.social.florentine_families_graph

networkx.generators.social.**florentine_families_graph**()
Return Florentine families graph.

References

[R187]

CHAPTER

SEVEN

LINEAR ALGEBRA

7.1 Spectrum

Laplacian, adjacency matrix, and spectrum of graphs.

adj_matrix(G[, nodelist, weight])	Return adjacency matrix of G.
laplacian(G[, nodelist, weight])	Return the Laplacian matrix of G.
normalized_laplacian(G[, nodelist, weight])	Return the normalized Laplacian matrix of G.
laplacian_spectrum(G[, weight])	Return eigenvalues of the Laplacian of G
adjacency_spectrum(G[, weight])	Return eigenvalues of the adjacency matrix of G.

7.1.1 networkx.linalg.spectrum.adj_matrix

networkx.linalg.spectrum.**adj_matrix**(*G*, *nodelist=None*, *weight='weight'*)

Return 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().

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** : numpy matrix

Adjacency matrix representation of G.

See Also:

to_numpy_matrix, to_dict_of_dicts

Notes

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, the edges weights are summed. See to_numpy_matrix for other options.

7.1.2 networkx.linalg.spectrum.laplacian

networkx.linalg.spectrum.**laplacian**(*G*, *nodelist=None*, *weight='weight'*)

Return 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** : NumPy array

Laplacian of G.

See Also:

to_numpy_matrix, normalized_laplacian

Notes

For MultiGraph/MultiDiGraph, the edges weights are summed. See to_numpy_matrix for other options.

7.1.3 networkx.linalg.spectrum.normalized_laplacian

networkx.linalg.spectrum.**normalized_laplacian**(*G*, *nodelist=None*, *weight='weight'*)

Return the normalized Laplacian matrix of G.

The normalized graph Laplacian is the matrix NL=D^(-1/2) L D^(-1/2) L is the graph Laplacian 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** : NumPy array

Normalized Laplacian of G.

See Also:

laplacian

Notes

For MultiGraph/MultiDiGraph, the edges weights are summed. See to_numpy_matrix for other options.

References

[R190]

7.1.4 networkx.linalg.spectrum.laplacian_spectrum

networkx.linalg.spectrum.**laplacian_spectrum**(*G*, *weight='weight'*)

Return 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

Notes

For MultiGraph/MultiDiGraph, the edges weights are summed. See to_numpy_matrix for other options.

7.1.5 networkx.linalg.spectrum.adjacency_spectrum

networkx.linalg.spectrum.**adjacency_spectrum**(*G*, *weight='weight'*)

Return 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:

adj_matrix

Notes

For MultiGraph/MultiDiGraph, the edges weights are summed. See to_numpy_matrix for other options.

7.2 Attribute Matrices

Functions for constructing matrix-like objects from graph attributes.

attr_matrix(G[, edge_attr, node_attr, ...])	Returns a NumPy matrix using attributes from G.
attr_sparse_matrix(G[, edge_attr, ...])	Returns a SciPy sparse matrix using attributes from G.

7.2.1 networkx.linalg.attrmatrix.attr_matrix

networkx.linalg.attrmatrix.**attr_matrix**(*G*, *edge_attr=None*, *node_attr=None*, *normalized=False*, *rc_order=None*, *dtype=None*, *order=None*)

Returns a NumPy matrix 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** : NumPy matrix

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.

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])
matrix([[ 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])
matrix([[ 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.node[0]['color'] = 'red'
>>> G.node[1]['color'] = 'red'
>>> G.node[2]['color'] = 'blue'
>>> rc = ['red', 'blue']
>>> nx.attr_matrix(G, node_attr='color', normalized=True, rc_order=rc)
matrix([[ 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)
matrix([[ 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

7.2.2 networkx.linalg.attrmatrix.attr_sparse_matrix

networkx.linalg.attrmatrix.**attr_sparse_matrix**(*G*, *edge_attr=None*, *node_attr=None*, *normalized=False*, *rc_order=None*, *dtype=None*)

Returns a SciPy sparse matrix 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 matrix

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.todense()
matrix([[ 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.todense()
matrix([[ 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.node[0]['color'] = 'red'
>>> G.node[1]['color'] = 'red'
>>> G.node[2]['color'] = 'blue'
>>> rc = ['red', 'blue']
>>> M = nx.attr_sparse_matrix(G, node_attr='color',                    normaliz
>>> M.todense()
matrix([[ 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_att
>>> M.todense()
matrix([[ 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

CHAPTER

EIGHT

CONVERTING TO AND FROM OTHER DATA FORMATS

8.1 To NetworkX Graph

This module provides functions to convert NetworkX graphs to and from other formats.

The preferred way of converting data to a NetworkX graph is through the graph constuctor. The constructor calls the to_networkx_graph() function which attempts to guess the input type and convert it automatically.

8.1.1 Examples

Create a 10 node random graph from a numpy matrix

```
>>> import numpy
>>> a=numpy.reshape(numpy.random.random_integers(0,1,size=100),(10,10))
>>> D=nx.DiGraph(a)
```

or equivalently

```
>>> D=nx.to_networkx_graph(a,create_using=nx.DiGraph())
```

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)
```

8.1.2 See Also

nx_pygraphviz, nx_pydot

to_networkx_graph(data[, create_using, ...])	Make a NetworkX graph from a known data structure.

8.1.3 networkx.convert.to_networkx_graph

networkx.convert.**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** : a object to be converted

Current known types are: any NetworkX graph dict-of-dicts dist-of-lists list of edges numpy matrix numpy ndarray scipy sparse matrix pygraphviz agraph

create_using : NetworkX graph

Use specified graph for result. Otherwise a new graph is created.

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.

8.2 Dictionaries

to_dict_of_dicts(G[, nodelist, edge_data])	Return adjacency representation of graph as a dictionary of dictionaries.
from_dict_of_dicts(d[, create_using, ...])	Return a graph from a dictionary of dictionaries.

8.2.1 networkx.convert.to_dict_of_dicts

networkx.convert.**to_dict_of_dicts**(*G*, *nodelist=None*, *edge_data=None*)

Return 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 : list, optional

If provided, the value of the dictionary will be set to edge_data for all edges. This is useful to make an adjacency matrix type representation with 1 as the edge data. If edgedata is None, the edgedata in G is used to fill the values. If G is a multigraph, the edgedata is a dict for each pair (u,v).

8.2.2 networkx.convert.from_dict_of_dicts

networkx.convert.**from_dict_of_dicts**(*d*, *create_using=None*, *multigraph_input=False*)

Return a graph from a dictionary of dictionaries.

Parameters **d** : dictionary of dictionaries

A dictionary of dictionaries adjacency representation.

create_using : NetworkX graph

Use specified graph for result. Otherwise a new graph is created.

multigraph_input : bool (default False)

When True, the values of the inner dict are assumed to be containers of edge data for multiple edges. Otherwise this routine assumes the edge data are singletons.

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

8.3 Lists

to_dict_of_lists(G[, nodelist])	Return adjacency representation of graph as a dictionary of lists.
from_dict_of_lists(d[, create_using])	Return a graph from a dictionary of lists.
to_edgelist(G[, nodelist])	Return a list of edges in the graph.
from_edgelist(edgelist[, create_using])	Return a graph from a list of edges.

8.3.1 networkx.convert.to_dict_of_lists

networkx.convert.**to_dict_of_lists**(*G*, *nodelist=None*)

Return 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.

8.3.2 networkx.convert.from_dict_of_lists

networkx.convert.**from_dict_of_lists**(*d*, *create_using=None*)

Return a graph from a dictionary of lists.

Parameters **d** : dictionary of lists

A dictionary of lists adjacency representation.

create_using : NetworkX graph

Use specified graph for result. Otherwise a new graph is created.

Examples

```
>>> dol= {0:[1]} # single edge (0,1)
>>> G=nx.from_dict_of_lists(dol)
```

or >>> G=nx.Graph(dol) # use Graph constructor

8.3.3 networkx.convert.to_edgelist

networkx.convert.**to_edgelist**(*G*, *nodelist=None*)

Return a list of edges in the graph.

Parameters **G** : graph

A NetworkX graph

nodelist : list

Use only nodes specified in nodelist

8.3.4 networkx.convert.from_edgelist

networkx.convert.**from_edgelist**(*edgelist*, *create_using=None*)

Return a graph from a list of edges.

Parameters **edgelist** : list or iterator

Edge tuples

create_using : NetworkX graph

Use specified graph for result. Otherwise a new graph is created.

Examples

```
>>> edgelist= [(0,1)] # single edge (0,1)
>>> G=nx.from_edgelist(edgelist)
```

or >>> G=nx.Graph(edgelist) # use Graph constructor

8.4 Numpy

to_numpy_matrix(G[, nodelist, dtype, order, ...])	Return the graph adjacency matrix as a NumPy matrix.
to_numpy_recarray(G[, nodelist, dtype, order])	Return the graph adjacency matrix as a NumPy recarray.
from_numpy_matrix(A[, create_using])	Return a graph from numpy matrix.

8.4.1 networkx.convert.to_numpy_matrix

networkx.convert.**to_numpy_matrix**(*G*, *nodelist=None*, *dtype=None*, *order=None*, *multigraph_weight=<built-in function sum>*, *weight='weight'*)

Return the graph adjacency matrix as a NumPy matrix.

Parameters **G** : graph

The NetworkX graph used to construct the NumPy 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 single NumPy data type used to initialize the array. This must be a simple type such as int or numpy.float64 and not a compound data type (see to_numpy_recarray) If None, then 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. If None, then the NumPy default is used.

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, optional :

Edge data key corresponding to the edge weight.

Returns **M** : NumPy matrix

Graph adjacency matrix.

See Also:

`to_numpy_recarray`, `from_numpy_matrix`

Notes

The matrix entries are assigned with weight edge attribute. When an edge does not have the weight attribute, the value of the entry is 1. For multiple edges, the values of the entries are the sums of the edge attributes for each edge.

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 $nodelist$.

Examples

```
>>> G = nx.MultiDiGraph()
>>> G.add_edge(0,1,weight=2)
>>> G.add_edge(1,0)
>>> G.add_edge(2,2,weight=3)
>>> G.add_edge(2,2)
>>> nx.to_numpy_matrix(G, nodelist=[0,1,2])
matrix([[ 0.,  2.,  0.],
        [ 1.,  0.,  0.],
        [ 0.,  0.,  4.]])
```

8.4.2 networkx.convert.to_numpy_recarray

networkx.convert.**to_numpy_recarray**(*G, nodelist=None, dtype=[('weight', <type 'float'>)], order=None*)

Return the graph adjacency matrix as a NumPy recarray.

Parameters **G** : graph

The NetworkX graph used to construct the NumPy 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 named dtype used to initialize the NumPy recarray. The data type names are assumed to be keys in the graph edge attribute dictionary.

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.

Returns **M** : NumPy recarray

The graph with specified edge data as a Numpy recarray

Notes

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 $nodelist$.

Examples

```
>>> G = nx.Graph()
>>> G.add_edge(1,2,weight=7.0,cost=5)
>>> A=nx.to_numpy_recarray(G,dtype=[('weight',float),('cost',int)])
>>> print(A.weight)
[[ 0.  7.]
 [ 7.  0.]]
>>> print(A.cost)
[[0 5]
 [5 0]]
```

8.4.3 networkx.convert.from_numpy_matrix

networkx.convert.**from_numpy_matrix**(*A, create_using=None*)

Return a graph from numpy matrix.

The numpy matrix is interpreted as an adjacency matrix for the graph.

Parameters **A** : numpy matrix

An adjacency matrix representation of a graph

create_using : NetworkX graph

Use specified graph for result. The default is Graph()

See Also:

to_numpy_matrix, to_numpy_recarray

Notes

If the numpy matrix has a single data type for each matrix entry it will be converted to an appropriate Python data type.

If the numpy matrix 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
>>> A=numpy.matrix([[1,1],[2,1]])
>>> G=nx.from_numpy_matrix(A)
```

User defined compound data type on edges:

```
>>> import numpy
>>> dt=[('weight',float),('cost',int)]
>>> A=numpy.matrix([[(1.0,2)]],dtype=dt)
>>> G=nx.from_numpy_matrix(A)
>>> G.edges(data=True)
[(0, 0, {'cost': 2, 'weight': 1.0})]
```

8.5 Scipy

to_scipy_sparse_matrix(G[, nodelist, dtype])	Return the graph adjacency matrix as a SciPy sparse matrix.
from_scipy_sparse_matrix(A[, create_using])	Return a graph from scipy sparse matrix adjacency list.

8.5.1 networkx.convert.to_scipy_sparse_matrix

networkx.convert.**to_scipy_sparse_matrix**(*G*, *nodelist=None*, *dtype=None*)

Return the graph adjacency matrix as a SciPy sparse matrix.

Parameters **G** : graph

The NetworkX graph used to construct the NumPy 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.

Returns **M** : SciPy sparse matrix

Graph adjacency matrix.

Notes

The matrix entries are populated using the 'weight' edge attribute. When an edge does not have the 'weight' 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 matrix is built from the subgraph of G that is induced by the nodes in $nodelist$.

Uses lil_matrix format. To convert to other formats see the documentation for scipy.sparse.

Examples

```
>>> G = nx.MultiDiGraph()
>>> G.add_edge(0,1,weight=2)
>>> G.add_edge(1,0)
>>> G.add_edge(2,2,weight=3)
>>> G.add_edge(2,2)
>>> S = nx.to_scipy_sparse_matrix(G, nodelist=[0,1,2])
>>> S.todense()
matrix([[ 0.,  2.,  0.],
        [ 1.,  0.,  0.],
        [ 0.,  0.,  4.]])
```

8.5.2 networkx.convert.from_scipy_sparse_matrix

networkx.convert.**from_scipy_sparse_matrix**(*A*, *create_using=None*)

Return a graph from scipy sparse matrix adjacency list.

Parameters **A** : scipy sparse matrix

An adjacency matrix representation of a graph

create_using : NetworkX graph

Use specified graph for result. The default is Graph()

Examples

```
>>> import scipy.sparse
>>> A=scipy.sparse.eye(2,2,1)
>>> G=nx.from_scipy_sparse_matrix(A)
```

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 networkx.readwrite.adjlist.read_adjlist

networkx.readwrite.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 container :

Use given NetworkX graph for holding nodes or edges.

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.

create_using: NetworkX graph container :

Use given NetworkX graph for holding nodes or edges.

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 is a NetworkX graph container. The default is 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 networkx.readwrite.adjlist.write_adjlist

networkx.readwrite.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

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 networkx.readwrite.adjlist.parse_adjlist

networkx.readwrite.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 container :

Use given NetworkX graph for holding nodes or edges.

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.

create_using: NetworkX graph container :

Use given NetworkX graph for holding nodes or edges.

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)
>>> G.nodes()
[1, 2, 3, 4, 5]
>>> G.edges()
[(1, 2), (1, 5), (2, 3), (2, 4), (3, 5)]
```

9.1.5 networkx.readwrite.adjlist.generate_adjlist

networkx.readwrite.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`

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 networkx.readwrite.multiline_adjlist.read_multiline_adjlist

networkx.readwrite.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 container :

Use given NetworkX graph for holding nodes or edges.

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.

create_using: NetworkX graph container :

Use given NetworkX graph for holding nodes or edges.

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 networkx.readwrite.multiline_adjlist.write_multiline_adjlist

networkx.readwrite.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

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 networkx.readwrite.multiline_adjlist.parse_multiline_adjlist

networkx.readwrite.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 container :

Use given NetworkX graph for holding nodes or edges.

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.

create_using: NetworkX graph container :

Use given NetworkX graph for holding nodes or edges.

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)
>>> G.nodes()
[1, 2, 3, 5]
>>> G.edges(data = True)
[(1, 2, {'name': 'Frodo', 'weight': 3}), (1, 3, {}), (2, 5, {'name': 'Saruman', 'weight': 6})]
```

9.2.5 networkx.readwrite.multiline_adjlist.generate_multiline_adjlist

networkx.readwrite.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 networkx.readwrite.edgelist.read_edgelist

networkx.readwrite.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 write. 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

```
>>> 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)
```

```
>>> 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'
>>> n=open('test.edgelist','w').write(textline)
>>> G = nx.read_edgelist('test.edgelist', nodetype=int, data=(('weight',float),))
>>> G.nodes()
[1, 2]
>>> G.edges(data = True)
[(1, 2, {'weight': 3.0})]
```

See parse_edgelist() for more examples of formatting.

9.3.3 networkx.readwrite.edgelist.write_edgelist

networkx.readwrite.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:

write_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 networkx.readwrite.edgelist.read_weighted_edgelist

networkx.readwrite.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 write. 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

encoding: string, optional :

Specify which encoding to use when reading file.

Returns **G** : graph

A networkx Graph or other type specified with create_using

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 networkx.readwrite.edgelist.write_weighted_edgelist

networkx.readwrite.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, write_weighted_edgelist

Examples

```
>>> G=nx.Graph()
>>> G.add_edge(1,2,weight=7)
>>> nx.write_weighted_edgelist(G, 'test.weighted.edgelist')
```

9.3.6 networkx.readwrite.edgelist.generate_edgelist

networkx.readwrite.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 networkx.readwrite.edgelist.parse_edgelist

networkx.readwrite.edgelist.**parse_edgelist**(*lines*, *comments='#'*, *delimiter=None*, *create_using=None*, *nodetype=None*, *data=True*)

Parse lines of an edge list representation of a graph.

Returns **G: NetworkX Graph** :

The graph corresponding to lines

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.

create_using: NetworkX graph container, optional :

Use given NetworkX graph for holding nodes or edges.

nodetype : Python type, optional

Convert nodes to this type.

comments : string, optional

Marker for comment lines

delimiter : string, optional

Separator for node labels

create_using: NetworkX graph container :

Use given NetworkX graph for holding nodes or edges.

See Also:

`read_weighted_edgelist`

Examples

Edgelist with no data:

```
>>> lines = ["1 2",
...          "2 3",
...          "3 4"]
>>> G = nx.parse_edgelist(lines, nodetype = int)
>>> G.nodes()
[1, 2, 3, 4]
>>> 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)
>>> G.nodes()
[1, 2, 3, 4]
>>> 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),))
>>> G.nodes()
[1, 2, 3, 4]
>>> 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.

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 unidirected edges together).

9.4.1 Format

GEXF is an XML format. See http://gexf.net/format/schema.html for the specification and http://gexf.net/format/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.
relabel_gexf_graph(G)	Relabel graph using "label" node keyword for node label.

9.4.2 networkx.readwrite.gexf.read_gexf

networkx.readwrite.gexf.**read_gexf**(*path*, *node_type=<type 'str'>*, *relabel=False*, *version='1.1draft'*)

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" [R191].

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

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.

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 unidirected edges together).

References

[R191]

9.4.3 networkx.readwrite.gexf.write_gexf

networkx.readwrite.gexf.**write_gexf**(*G*, *path*, *encoding='utf-8'*, *prettyprint=True*, *version='1.1draft'*)

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" [R192].

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.

Notes

This implementation does not support mixed graphs (directed and unidirected 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

[R192]

Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_gexf(G, "test.gexf")
```

9.4.4 networkx.readwrite.gexf.relabel_gexf_graph

networkx.readwrite.gexf.**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 relabed nodes

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 G>raph 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."

See http://www.infosun.fim.uni-passau.de/Graphlet/GML/gml-tr.html

Requires pyparsing: http://pyparsing.wikispaces.com/

9.5.1 Format

See http://www.infosun.fim.uni-passau.de/Graphlet/GML/gml-tr.html for format specification.

Example graphs in GML format: http://www-personal.umich.edu/~mejn/netdata/

read_gml(path[, encoding, relabel])	Read graph in GML format from path.
write_gml(G, path)	Write the graph G in GML format to the file or file handle path.
parse_gml(lines[, relabel])	Parse GML graph from a string or iterable.
generate_gml(G)	Generate a single entry of the graph G in GML format.

9.5.2 networkx.readwrite.gml.read_gml

networkx.readwrite.gml.**read_gml**(*path*, *encoding='UTF-8'*, *relabel=False*)

Read graph in GML format from path.

Parameters **path** : filename or filehandle

The filename or filehandle to read from.

encoding : string, optional

Text encoding.

relabel : bool, optional

If True use the GML node label attribute for node names otherwise use the node id.

Returns **G** : MultiGraph or MultiDiGraph

Raises **ImportError** :

If the pyparsing module is not available.

See Also:

write_gml, parse_gml

Notes

Requires pyparsing: http://pyparsing.wikispaces.com/

References

GML specification: http://www.infosun.fim.uni-passau.de/Graphlet/GML/gml-tr.html

Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_gml(G,'test.gml')
>>> H=nx.read_gml('test.gml')
```

9.5.3 networkx.readwrite.gml.write_gml

networkx.readwrite.gml.**write_gml**(*G*, *path*)

Write the graph G in GML format to the file or file handle path.

Parameters **path** : filename or filehandle

The filename or filehandle to write. Filenames ending in .gz or .gz2 will be compressed.

See Also:

read_gml, parse_gml

Notes

GML specifications indicate that the file should only use 7bit ASCII text encoding.iso8859-1 (latin-1).

This implementation does not support all Python data types as GML data. Nodes, node attributes, edge attributes, and graph attributes must be either dictionaries or single stings or numbers. If they are not an attempt is made to represent them as strings. For example, a list as edge data G[1][2]['somedata']=[1,2,3], will be represented in the GML file as:

```
edge [
  source 1
  target 2
  somedata "[1, 2, 3]"
]
```

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")
```

9.5.4 networkx.readwrite.gml.parse_gml

networkx.readwrite.gml.**parse_gml**(*lines*, *relabel=True*)

Parse GML graph from a string or iterable.

Parameters **lines** : string or iterable

Data in GML format.

relabel : bool, optional

If True use the GML node label attribute for node names otherwise use the node id.

Returns **G** : MultiGraph or MultiDiGraph

Raises **ImportError** :

If the pyparsing module is not available.

See Also:

`write_gml`, `read_gml`

Notes

This stores nested GML attributes as dictionaries in the NetworkX graph, node, and edge attribute structures.

Requires pyparsing: http://pyparsing.wikispaces.com/

References

GML specification: http://www.infosun.fim.uni-passau.de/Graphlet/GML/gml-tr.html

9.5.5 networkx.readwrite.gml.generate_gml

`networkx.readwrite.gml.`**`generate_gml`**`(G)`

Generate a single entry of the graph G in GML format.

Parameters **G** : NetworkX graph

Returns **lines: string** :

Lines in GML format.

Notes

This implementation does not support all Python data types as GML data. Nodes, node attributes, edge attributes, and graph attributes must be either dictionaries or single stings or numbers. If they are not an attempt is made to represent them as strings. For example, a list as edge data G[1][2]['somedata']=[1,2,3], will be represented in the GML file as:

```
edge [
  source 1
  target 2
  somedata "[1, 2, 3]"
]
```

9.6 Pickle

Read and write NetworkX graphs as Python pickles.

"The pickle module implements a fundamental, but powerful algorithm for serializing and de-serializing a Python object structure. "Pickling" is the process whereby a Python object hierarchy is converted into a byte stream, and "unpickling" is the inverse operation, whereby a byte stream is converted back into an object hierarchy."

Note that NetworkX graphs can contain any hashable Python object as node (not just integers and strings). For arbitrary data types it may be difficult to represent the data as text. In that case using Python pickles to store the graph data can be used.

9.6.1 Format

See http://docs.python.org/library/pickle.html

`read_gpickle`(path)	Read graph object in Python pickle format.
`write_gpickle`(G, path)	Write graph in Python pickle format.

9.6.2 networkx.readwrite.gpickle.read_gpickle

networkx.readwrite.gpickle.**read_gpickle**(*path*)

Read graph object in Python pickle format.

Pickles are a serialized byte stream of a Python object [R193]. This format will preserve Python objects used as nodes or edges.

Parameters **path** : file or string

File or filename to write. Filenames ending in .gz or .bz2 will be uncompressed.

Returns **G** : graph

A NetworkX graph

References

[R193]

Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_gpickle(G,"test.gpickle")
>>> G=nx.read_gpickle("test.gpickle")
```

9.6.3 networkx.readwrite.gpickle.write_gpickle

networkx.readwrite.gpickle.**write_gpickle**(*G*, *path*)

Write graph in Python pickle format.

Pickles are a serialized byte stream of a Python object [R194]. This format will preserve Python objects used as nodes or edges.

Parameters **G** : graph

A NetworkX graph

path : file or string

File or filename to write. Filenames ending in .gz or .bz2 will be compressed.

References

[R194]

Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_gpickle(G,"test.gpickle")
```

9.7 GraphML

Read and write graphs in GraphML format.

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/

9.7.1 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, prettyprint])	Write G in GraphML XML format to path

9.7.2 networkx.readwrite.graphml.read_graphml

networkx.readwrite.graphml.**read_graphml**(*path*, *node_type=<type 'str'>*)

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

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 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 will can be read but the graphics information is discarded.

yEd compressed files ("file.graphmlz" extension) can be read by renaming the file to "file.graphml.gz".

9.7.3 networkx.readwrite.graphml.write_graphml

networkx.readwrite.graphml.**write_graphml**(*G*, *path*, *encoding='utf-8'*, *prettyprint=True*)

Write G in GraphML XML 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.

encoding : string (optional)

Encoding for text data.

prettyprint : bool (optional)

If True use line breaks and indenting in output XML.

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(G, "test.graphml")
```

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 networkx.readwrite.leda.read_leda

networkx.readwrite.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

[R196]

Examples

G=nx.read_leda('file.leda')

9.8.3 networkx.readwrite.leda.parse_leda

networkx.readwrite.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

[R195]

Examples

G=nx.parse_leda(string)

9.9 YAML

Read and write NetworkX graphs in YAML format.

"YAML is a data serialization format designed for human readability and interaction with scripting languages." See http://www.yaml.org for documentation.

9.9.1 Format

http://pyyaml.org/wiki/PyYAML

`read_yaml`(path)	Read graph in YAML format from path.
`write_yaml`(G, path, **kwds[, encoding])	Write graph G in YAML format to path.

9.9.2 networkx.readwrite.nx_yaml.read_yaml

networkx.readwrite.nx_yaml.**read_yaml**(*path*)

Read graph in YAML format from path.

YAML is a data serialization format designed for human readability and interaction with scripting languages [R198].

Parameters **path** : file or string

File or filename to read. Filenames ending in .gz or .bz2 will be uncompressed.

Returns **G** : NetworkX graph

References

[R198]

Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_yaml(G,'test.yaml')
>>> G=nx.read_yaml('test.yaml')
```

9.9.3 networkx.readwrite.nx_yaml.write_yaml

networkx.readwrite.nx_yaml.**write_yaml**(*G*, *path*, *encoding='UTF-8'*, ***kwds*)

Write graph G in YAML format to path.

YAML is a data serialization format designed for human readability and interaction with scripting languages [R199].

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 :

Specify which encoding to use when writing file.

References

[R199]

Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_yaml(G,'test.yaml')
```

9.10 SparseGraph6

Read graphs in graph6 and sparse6 format.

9.10.1 Format

"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." http://cs.anu.edu.au/~bdm/data/formats.html

See http://cs.anu.edu.au/~bdm/data/formats.txt for details.

read_graph6(path)	Read simple undirected graphs in graph6 format from path.
parse_graph6(str)	Read a simple undirected graph in graph6 format from string.
read_graph6_list(path)	Read simple undirected graphs in graph6 format from path.
read_sparse6(path)	Read simple undirected graphs in sparse6 format from path.
parse_sparse6(string)	Read undirected graph in sparse6 format from string.
read_sparse6_list(path)	Read undirected graphs in sparse6 format from path.

9.10.2 networkx.readwrite.sparsegraph6.read_graph6

networkx.readwrite.sparsegraph6.**read_graph6**(*path*)

Read simple undirected graphs in graph6 format from path.

Returns a single Graph.

9.10.3 networkx.readwrite.sparsegraph6.parse_graph6

networkx.readwrite.sparsegraph6.**parse_graph6**(*str*)

Read a simple undirected graph in graph6 format from string.

Returns a single Graph.

9.10.4 networkx.readwrite.sparsegraph6.read_graph6_list

networkx.readwrite.sparsegraph6.**read_graph6_list**(*path*)

Read simple undirected graphs in graph6 format from path.

Returns a list of Graphs, one for each line in file.

9.10.5 networkx.readwrite.sparsegraph6.read_sparse6

networkx.readwrite.sparsegraph6.**read_sparse6**(*path*)
Read simple undirected graphs in sparse6 format from path.

Returns a single MultiGraph.

9.10.6 networkx.readwrite.sparsegraph6.parse_sparse6

networkx.readwrite.sparsegraph6.**parse_sparse6**(*string*)
Read undirected graph in sparse6 format from string.

Returns a MultiGraph.

9.10.7 networkx.readwrite.sparsegraph6.read_sparse6_list

networkx.readwrite.sparsegraph6.**read_sparse6_list**(*path*)
Read undirected graphs in sparse6 format from path.

Returns a list of MultiGraphs, one for each line in file.

9.11 Pajek

Read graphs in Pajek format.

This implementation handles directed and undirected graphs including those with self loops and parallel edges.

9.11.1 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.

9.11.2 networkx.readwrite.pajek.read_pajek

networkx.readwrite.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)
```

9.11.3 networkx.readwrite.pajek.write_pajek

networkx.readwrite.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.

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")
```

9.11.4 networkx.readwrite.pajek.parse_pajek

networkx.readwrite.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

9.12 GIS Shapefile

Generates a networkx.DiGraph from point and line shapefiles.

Point geometries are translated into nodes, lines into edges. Coordinate tuples are used as keys. Attributes are preserved, line geometries are simplified into start and end coordinates. Accepts a single shapefile or directory of many shapefiles.

"The Esri Shapefile or simply a shapefile is a popular geospatial vector data format for geographic information systems software. It is developed and regulated by Esri as a (mostly) open specification for data interoperability among Esri and other software products." See http://en.wikipedia.org/wiki/Shapefile for additional information.

read_shp(path)	Generate a directed graph from shapefiles.

9.12.1 networkx.readwrite.nx_shp.read_shp

networkx.readwrite.nx_shp.**read_shp**(*path*)

Generate a directed graph from shapefiles.

"The Esri Shapefile or simply a shapefile is a popular geospatial vector data format for geographic information systems software [R197]."

Point geometries are translated into nodes, lines into edges. Coordinate tuples are used as keys. Attributes are preserved, line geometries are simplified into start and end coordinates.

Parameters **path** : string

File name or directory name.

Returns **G** : NetworkX DiGraph

Notes

Uses Python bindings for OGR in the GDAL library, http://www.gdal.org. Available for Linux in the python-gdal package.

References

[R197]

Examples

G=nx.read_shp('test.shp')

CHAPTER

TEN

DRAWING

10.1 Matplotlib

Draw networks with matplotlib (pylab).

10.1.1 See Also

matplotlib: http://matplotlib.sourceforge.net/

pygraphviz: http://networkx.lanl.gov/pygraphviz/

draw(G, **kwds[, pos, ax, hold])	Draw the graph G with Matplotlib (pylab).
draw_networkx(G, **kwds[, pos, with_labels])	Draw the graph G using Matplotlib.
draw_networkx_nodes(G, pos, **kwds[, ...])	Draw the nodes of the graph G.
draw_networkx_edges(G, pos, **kwds[, ...])	Draw the edges of the graph G.
draw_networkx_labels(G, pos, **kwds[, ...])	Draw node labels on the graph G.
draw_networkx_edge_labels(G, pos, **kwds[, ...])	Draw edge labels.
draw_circular(G, **kwargs)	Draw the graph G with a circular layout.
draw_random(G, **kwargs)	Draw the graph G with a random layout.
draw_spectral(G, **kwargs)	Draw the graph G with a spectral layout.
draw_spring(G, **kwargs)	Draw the graph G with a spring layout.
draw_shell(G, **kwargs)	Draw networkx graph with shell layout.
draw_graphviz(G, **kwargs[, prog])	Draw networkx graph with graphviz layout.

10.1.2 networkx.drawing.nx_pylab.draw

networkx.drawing.nx_pylab.**draw**(*G*, *pos=None*, *ax=None*, *hold=None*, ***kwds*)

Draw the graph G with Matplotlib (pylab).

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.layout for functions that compute node positions.

ax : Matplotlib Axes object, optional

Draw the graph in specified Matplotlib axes.

hold: bool, optional :

Set the Matplotlib hold state. If True subsequent draw commands will be added to the current 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.

Good alternatives are:

With pylab:

```
>>> import pylab as P #
>>> import networkx as nx
>>> G=nx.dodecahedral_graph()
>>> nx.draw(G)   # networkx draw()
>>> P.draw()     # pylab draw()
```

With pyplot

```
>>> import matplotlib.pyplot as plt
>>> import networkx as nx
>>> G=nx.dodecahedral_graph()
>>> nx.draw(G)   # networkx draw()
>>> plt.draw()   # pyplot draw()
```

Also see the NetworkX drawing examples at http://networkx.lanl.gov/gallery.html

Examples

```
>>> G=nx.dodecahedral_graph()
>>> nx.draw(G)
>>> nx.draw(G,pos=nx.spring_layout(G)) # use spring layout
```

10.1.3 networkx.drawing.nx_pylab.draw_networkx

networkx.drawing.nx_pylab.**draw_networkx**(*G*, *pos=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.layout for functions that compute node positions.

ax : Matplotlib Axes object, optional

Draw the graph in the specified Matplotlib axes.

with_labels: bool, optional :

Set to True (default) to draw labels on the nodes.

nodelist: list, optional :

Draw only specified nodes (default G.nodes())

edgelist: list :

Draw only specified edges(default=G.edges())

node_size: scalar or array :

Size of nodes (default=300). If an array is specified it must be the same length as nodelist.

node_color: color string, or array of floats :

Node color. Can be a single color format string (default='r'), or a sequence of colors with the same length as nodelist. 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 :

The shape of the node. Specification is as matplotlib.scatter marker, one of 'so^>v<dph8' (default='o').

alpha: float :

The node transparency (default=1.0)

cmap: Matplotlib colormap :

Colormap for mapping intensities of nodes (default=None)

vmin,vmax: floats :

Minimum and maximum for node colormap scaling (default=None)

width': float :

Line width of edges (default =1.0)

edge_color: color string, or array of floats :

Edge color. Can be a single color format string (default='r'), or a sequence of colors with the same length as edgelist. 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 :

Colormap for mapping intensities of edges (default=None)

edge_vmin,edge_vmax: floats :

Minimum and maximum for edge colormap scaling (default=None)

style: string :

Edge line style (default='solid') (solid|dashed|dotted,dashdot)

labels: dictionary :

Node labels in a dictionary keyed by node of text labels (default=None)

font_size: int :

Font size for text labels (default=12)

font_color: string :

Font color string (default='k' black)

font_weight: string :

Font weight (default='normal')

font_family: string :

Font family (default='sans-serif')

See Also:

draw, draw_networkx_nodes, draw_networkx_edges, draw_networkx_labels, draw_networkx_edge_labels

Examples

```
>>> G=nx.dodecahedral_graph()
>>> nx.draw(G)
>>> nx.draw(G,pos=nx.spring_layout(G)) # use spring layout

>>> import pylab
>>> limits=pylab.axis('off') # turn of axis
```

Also see the NetworkX drawing examples at http://networkx.lanl.gov/gallery.html

10.1.4 networkx.drawing.nx_pylab.draw_networkx_nodes

networkx.drawing.nx_pylab.**draw_networkx_nodes**(*G*, *pos*, *nodelist=None*, *node_size=300*, *node_color='r'*, *node_shape='o'*, *alpha=1.0*, *cmap=None*, *vmin=None*, *vmax=None*, *ax=None*, *linewidths=None*, ***kwds*)

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. If not specified a spring layout positioning will be computed. See networkx.layout for functions that compute node positions.

ax : Matplotlib Axes object, optional

Draw the graph in the specified Matplotlib axes.

nodelist: list, optional :

Draw only specified nodes (default G.nodes())

edgelist: list :

Draw only specified edges(default=G.edges())

node_size: scalar or array :

Size of nodes (default=300). If an array is specified it must be the same length as nodelist.

node_color: color string, or array of floats :

Node color. Can be a single color format string (default='r'), or a sequence of colors with the same length as nodelist. 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 :

The shape of the node. Specification is as matplotlib.scatter marker, one of 'so^>v<dph8' (default='o').

alpha: float :

The node transparency (default=1.0)

cmap: Matplotlib colormap :

Colormap for mapping intensities of nodes (default=None)

vmin,vmax: floats :

Minimum and maximum for node colormap scaling (default=None)

width': float :

Line width of edges (default =1.0)

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 http://networkx.lanl.gov/gallery.html

10.1.5 networkx.drawing.nx_pylab.draw_networkx_edges

networkx.drawing.nx_pylab.**draw_networkx_edges**(*G*, *pos*, *edgelist=None*, *width=1.0*, *edge_color='k'*, *style='solid'*, *alpha=None*, *edge_cmap=None*, *edge_vmin=None*, *edge_vmax=None*, *ax=None*, *arrows=True*, ***kwds*)

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. If not specified a spring layout positioning will be computed. See networkx.layout for functions that compute node positions.

ax : Matplotlib Axes object, optional

Draw the graph in the specified Matplotlib axes.

alpha: float :

The edge transparency (default=1.0)

width': float :

Line width of edges (default =1.0)

edge_color: color string, or array of floats :

Edge color. Can be a single color format string (default='r'), or a sequence of colors with the same length as edgelist. 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 :

Colormap for mapping intensities of edges (default=None)

edge_vmin,edge_vmax: floats :

Minimum and maximum for edge colormap scaling (default=None)

style: string :

Edge line style (default='solid') (solid|dashed|dotted,dashdot)

See Also:

draw, draw_networkx, draw_networkx_nodes, draw_networkx_labels, draw_networkx_edge_labels

Notes

For directed graphs, "arrows" (actually just thicker stubs) are drawn at the head end. Arrows can be turned off with keyword arrows=False. Yes, it is ugly but drawing proper arrows with Matplotlib this way is tricky.

Examples

```
>>> G=nx.dodecahedral_graph()
>>> edges=nx.draw_networkx_edges(G,pos=nx.spring_layout(G))
```

Also see the NetworkX drawing examples at http://networkx.lanl.gov/gallery.html

10.1.6 networkx.drawing.nx_pylab.draw_networkx_labels

networkx.drawing.nx_pylab.**draw_networkx_labels**(*G*, *pos*, *labels=None*, *font_size=12*, *font_color='k'*, *font_family='sans-serif'*, *font_weight='normal'*, *alpha=1.0*, *ax=None*, ***kwds*)

Draw node labels on the graph G.

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.layout for functions that compute node positions.

ax : Matplotlib Axes object, optional

Draw the graph in the specified Matplotlib axes.

alpha: float :

The text transparency (default=1.0)

labels: dictionary :

Node labels in a dictionary keyed by node of text labels (default=None)

font_size: int :

Font size for text labels (default=12)

font_color: string :

Font color string (default='k' black)

font_weight: string :

Font weight (default='normal')

font_family: string :

Font family (default='sans-serif')

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 http://networkx.lanl.gov/gallery.html

10.1.7 networkx.drawing.nx_pylab.draw_networkx_edge_labels

networkx.drawing.nx_pylab.**draw_networkx_edge_labels**(*G*, *pos*, *edge_labels=None*, *font_size=10*, *font_color='k'*, *font_family='sans-serif'*, *font_weight='normal'*, *alpha=1.0*, *bbox=None*, *ax=None*, *rotate=True*, ***kwds*)

Draw edge labels.

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.layout for functions that compute node positions.

ax : Matplotlib Axes object, optional

Draw the graph in the specified Matplotlib axes.

alpha: float :

The text transparency (default=1.0)

labels: dictionary :

Node labels in a dictionary keyed by edge two-tuple of text labels (default=None), Only labels for the keys in the dictionary are drawn.

font_size: int :

Font size for text labels (default=12)

font_color: string :

Font color string (default='k' black)

font_weight: string :

Font weight (default='normal')

font_family: string :

Font family (default='sans-serif')

bbox: Matplotlib bbox :

Specify text box shape and colors.

clip_on: bool :

Turn on clipping at axis boundaries (default=True)

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 http://networkx.lanl.gov/gallery.html

10.1.8 networkx.drawing.nx_pylab.draw_circular

networkx.drawing.nx_pylab.**draw_circular**(*G*, ***kwargs*)
Draw the graph G with a circular layout.

10.1.9 networkx.drawing.nx_pylab.draw_random

networkx.drawing.nx_pylab.**draw_random**(*G*, ***kwargs*)
Draw the graph G with a random layout.

10.1.10 networkx.drawing.nx_pylab.draw_spectral

networkx.drawing.nx_pylab.**draw_spectral**(*G*, ***kwargs*)
Draw the graph G with a spectral layout.

10.1.11 networkx.drawing.nx_pylab.draw_spring

networkx.drawing.nx_pylab.**draw_spring**(*G*, ***kwargs*)
Draw the graph G with a spring layout.

10.1.12 networkx.drawing.nx_pylab.draw_shell

networkx.drawing.nx_pylab.**draw_shell**(*G*, ***kwargs*)
Draw networkx graph with shell layout.

10.1.13 networkx.drawing.nx_pylab.draw_graphviz

networkx.drawing.nx_pylab.**draw_graphviz**(*G*, *prog='neato'*, ***kwargs*)
Draw networkx graph with graphviz layout.

10.2 Graphviz AGraph (dot)

Interface to pygraphviz AGraph class.

10.2.1 Examples

```
>>> G=nx.complete_graph(5)
>>> A=nx.to_agraph(G)
>>> H=nx.from_agraph(A)
```

10.2.2 See Also

Pygraphviz: http://networkx.lanl.gov/pygraphviz

from_agraph(A[, create_using])	Return a NetworkX Graph or DiGraph from a PyGraphviz graph.
to_agraph(N)	Return 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)	Return 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 networkx.drawing.nx_agraph.from_agraph

networkx.drawing.nx_agraph.**from_agraph**(*A*, *create_using=None*)

Return a NetworkX Graph or DiGraph from a PyGraphviz graph.

Parameters **A** : PyGraphviz AGraph

A graph created with PyGraphviz

create_using : NetworkX graph class instance

The output is created using the given graph class instance

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.to_agraph(K5)
>>> G=nx.from_agraph(A)
>>> G=nx.from_agraph(A)
```

10.2.4 networkx.drawing.nx_agraph.to_agraph

networkx.drawing.nx_agraph.**to_agraph**(*N*)

Return 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.to_agraph(K5)
```

10.2.5 networkx.drawing.nx_agraph.write_dot

networkx.drawing.nx_agraph.**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.

10.2.6 networkx.drawing.nx_agraph.read_dot

networkx.drawing.nx_agraph.**read_dot**(*path*)
Return a NetworkX graph from a dot file on path.

Parameters **path** : file or string

File name or file handle to read.

10.2.7 networkx.drawing.nx_agraph.graphviz_layout

networkx.drawing.nx_agraph.**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

Dictionary of x,y, positions keyed by node.

Notes

This is a wrapper for pygraphviz_layout.

Examples

```
>>> G=nx.petersen_graph()
>>> pos=nx.graphviz_layout(G)
>>> pos=nx.graphviz_layout(G,prog='dot')
```

10.2.8 networkx.drawing.nx_agraph.pygraphviz_layout

networkx.drawing.nx_agraph.**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 : dictionary

Dictionary of x,y, positions keyed by node.

Examples

```
>>> G=nx.petersen_graph()
>>> pos=nx.graphviz_layout(G)
>>> pos=nx.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_pygraphviz can be used to interface with graphviz.

10.3.1 See Also

Pydot: http://www.dkbza.org/pydot.html Graphviz: http://www.research.att.com/sw/tools/graphviz/ DOT Language: http://www.graphviz.org/doc/info/lang.html

from_pydot(P)	Return a NetworkX graph from a Pydot graph.
to_pydot(N[, strict])	Return 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)	Return a NetworkX MultiGraph or MultiDiGraph from a dot file on path.
graphviz_layout(G, **kwds[, prog, root])	Create node positions using Pydot and Graphviz.
pydot_layout(G, **kwds[, prog, root])	Create node positions using Pydot and Graphviz.

10.3.2 networkx.drawing.nx_pydot.from_pydot

networkx.drawing.nx_pydot.**from_pydot**(*P*)

Return 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.to_pydot(K5)
>>> G=nx.from_pydot(A) # return MultiGraph
>>> G=nx.Graph(nx.from_pydot(A)) # make a Graph instead of MultiGraph
```

10.3.3 networkx.drawing.nx_pydot.to_pydot

networkx.drawing.nx_pydot.**to_pydot**(*N*, *strict=True*)

Return 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.to_pydot(K5)
```

10.3.4 networkx.drawing.nx_pydot.write_dot

networkx.drawing.nx_pydot.**write_dot**(*G*, *path*)

Write NetworkX graph G to Graphviz dot format on path.

Path can be a string or a file handle.

10.3.5 networkx.drawing.nx_pydot.read_dot

networkx.drawing.nx_pydot.**read_dot**(*path*)

Return a NetworkX MultiGraph or MultiDiGraph from a dot file on path.

Parameters **path** : filename or file handle

Returns **G** : NetworkX multigraph

A MultiGraph or MultiDiGraph.

Notes

Use G=nx.Graph(nx.read_dot(path)) to return a Graph instead of a MultiGraph.

10.3.6 networkx.drawing.nx_pydot.graphviz_layout

networkx.drawing.nx_pydot.**graphviz_layout**(*G*, *prog='neato'*, *root=None*, ***kwds*)

Create node positions using Pydot and Graphviz.

Returns a dictionary of positions keyed by node.

Notes

This is a wrapper for pydot_layout.

Examples

```
>>> G=nx.complete_graph(4)
>>> pos=nx.graphviz_layout(G)
>>> pos=nx.graphviz_layout(G,prog='dot')
```

10.3.7 networkx.drawing.nx_pydot.pydot_layout

networkx.drawing.nx_pydot.**pydot_layout**(*G*, *prog='neato'*, *root=None*, ***kwds*)

Create node positions using Pydot and Graphviz.

Returns a dictionary of positions keyed by node.

Examples

```
>>> G=nx.complete_graph(4)
>>> pos=nx.pydot_layout(G)
>>> pos=nx.pydot_layout(G,prog='dot')
```

10.4 Graph Layout

Node positioning algorithms for graph drawing.

circular_layout(G[, dim, scale])	Position nodes on a circle.
random_layout(G[, dim])	Position nodes uniformly at random in the unit square.
shell_layout(G[, nlist, dim, scale])	Position nodes in concentric circles.
spring_layout(G[, dim, pos, fixed, ...])	Position nodes using Fruchterman-Reingold force-directed algorithm.
spectral_layout(G[, dim, weighted, scale])	Position nodes using the eigenvectors of the graph Laplacian.

10.4.1 networkx.drawing.layout.circular_layout

networkx.drawing.layout.**circular_layout**(*G*, *dim=2*, *scale=1*)

Position nodes on a circle.

Parameters **G** : NetworkX graph

dim : int

Dimension of layout, currently only dim=2 is supported

scale : float

Scale factor for positions

Returns **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.path_graph(4)
>>> pos=nx.circular_layout(G)
```

10.4.2 networkx.drawing.layout.random_layout

networkx.drawing.layout.**random_layout**(*G*, *dim=2*)

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

A position will be assigned to every node in G.

dim : int

Dimension of layout.

Returns **dict :** :

A dictionary of positions keyed by node

Examples

```
>>> G = nx.lollipop_graph(4, 3)
>>> pos = nx.random_layout(G)
```

10.4.3 networkx.drawing.layout.shell_layout

networkx.drawing.layout.**shell_layout**(*G*, *nlist=None*, *dim=2*, *scale=1*)

Position nodes in concentric circles.

Parameters **G** : NetworkX graph

nlist : list of lists

List of node lists for each shell.

dim : int

Dimension of layout, currently only dim=2 is supported

scale : float

Scale factor for positions

Returns **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.path_graph(4)
>>> shells=[[0],[1,2,3]]
>>> pos=nx.shell_layout(G,shells)
```

10.4.4 networkx.drawing.layout.spring_layout

networkx.drawing.layout.**spring_layout**(*G*, *dim=2*, *pos=None*, *fixed=None*, *iterations=50*, *weighted=True*, *scale=1*)

Position nodes using Fruchterman-Reingold force-directed algorithm.

Parameters **G** : NetworkX graph

dim : int

Dimension of layout

pos : dict

Initial positions for nodes as a dictionary with node as keys and values as a list or tuple.

fixed : list

Nodes to keep fixed at initial position.

iterations : int

Number of iterations of spring-force relaxation

weighted : boolean

If True, use edge weights in layout

scale : float

Scale factor for positions

Returns **dict :** :

A dictionary of positions keyed by node

Examples

```
>>> G=nx.path_graph(4)
>>> pos=nx.spring_layout(G)
```

The same using longer function name >>> pos=nx.fruchterman_reingold_layout(G)

10.4.5 networkx.drawing.layout.spectral_layout

networkx.drawing.layout.**spectral_layout**(*G*, *dim=2*, *weighted=True*, *scale=1*)

Position nodes using the eigenvectors of the graph Laplacian.

Parameters **G** : NetworkX graph

dim : int

Dimension of layout

weighted : boolean

If True, use edge weights in layout

scale : float

Scale factor for positions

Returns **dict :** :

A dictionary of positions keyed by node

Notes

Directed graphs will be considered as unidirected 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)
```

CHAPTER

ELEVEN

EXCEPTIONS

Base exceptions and errors for NetworkX.

class networkx.**NetworkXException**
Base class for exceptions in NetworkX.

class networkx.**NetworkXError**
Exception for a serious error in NetworkX

class networkx.**NetworkXPointlessConcept**
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.

class networkx.**NetworkXAlgorithmError**
Exception for unexpected termination of algorithms.

class networkx.**NetworkXUnfeasible**
Exception raised by algorithms trying to solve a problem instance that has no feasible solution.

class networkx.**NetworkXNoPath**
Exception for algorithms that should return a path when running on graphs where such a path does not exist.

class networkx.**NetworkXUnbounded**
Exception raised by algorithms trying to solve a maximization or a minimization problem instance that is unbounded.

CHAPTER

TWELVE

UTILITIES

Helpers for NetworkX.

These are not imported into the base networkx namespace but can be accessed, for example, as

```
>>> import networkx
>>> networkx.utils.is_string_like('spam')
True
```

12.1 Helper functions

is_string_like(obj)	Check if obj is string.
flatten(obj[, result])	Return flattened version of (possibly nested) iterable object.
iterable(obj)	Return True if obj is iterable with a well-defined len().
is_list_of_ints(intlist)	Return True if list is a list of ints.
_get_fh(path[, mode])	Return a file handle for given path.

12.1.1 networkx.utils.is_string_like

networkx.utils.**is_string_like**(*obj*)

Check if obj is string.

12.1.2 networkx.utils.flatten

networkx.utils.**flatten**(*obj*, *result=None*)

Return flattened version of (possibly nested) iterable object.

12.1.3 networkx.utils.iterable

networkx.utils.**iterable**(*obj*)

Return True if obj is iterable with a well-defined len().

12.1.4 networkx.utils.is_list_of_ints

networkx.utils.**is_list_of_ints**(*intlist*)

Return True if list is a list of ints.

12.1.5 networkx.utils._get_fh

networkx.utils.**_get_fh**(*path*, *mode='r'*)
Return a file handle for given path.

Path can be a string or a file handle.

Attempt to uncompress/compress files ending in '.gz' and '.bz2'.

12.2 Data structures and Algorithms

UnionFind.union(*objects)	Find the sets containing the objects and merge them all.

12.2.1 networkx.utils.UnionFind.union

UnionFind.**union**(**objects*)
Find the sets containing the objects and merge them all.

12.3 Random sequence generators

pareto_sequence(n[, exponent])	Return sample sequence of length n from a Pareto distribution.
powerlaw_sequence(n[, exponent])	Return sample sequence of length n from a power law distribution.
uniform_sequence(n)	Return sample sequence of length n from a uniform distribution.
cumulative_distribution(distribution)	Return 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.

12.3.1 networkx.utils.pareto_sequence

networkx.utils.**pareto_sequence**(*n*, *exponent=1.0*)
Return sample sequence of length n from a Pareto distribution.

12.3.2 networkx.utils.powerlaw_sequence

networkx.utils.**powerlaw_sequence**(*n*, *exponent=2.0*)
Return sample sequence of length n from a power law distribution.

12.3.3 networkx.utils.uniform_sequence

networkx.utils.**uniform_sequence**(*n*)
Return sample sequence of length n from a uniform distribution.

12.3.4 networkx.utils.cumulative_distribution

networkx.utils.**cumulative_distribution**(*distribution*)
Return normalized cumulative distribution from discrete distribution.

12.3.5 networkx.utils.discrete_sequence

networkx.utils.**discrete_sequence**(*n*, *distribution=None*, *cdistribution=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

12.4 SciPy random sequence generators

scipy_pareto_sequence(n[, exponent])	Return sample sequence of length n from a Pareto distribution.
scipy_powerlaw_sequence(n[, exponent])	Return sample sequence of length n from a power law distribution.
scipy_poisson_sequence(n[, mu])	Return sample sequence of length n from a Poisson distribution.
scipy_uniform_sequence(n)	Return sample sequence of length n from a uniform distribution.
scipy_discrete_sequence(n[, distribution])	Return sample sequence of length n from a given discrete distribution.

12.4.1 networkx.utils.scipy_pareto_sequence

networkx.utils.**scipy_pareto_sequence**(*n*, *exponent=1.0*)
Return sample sequence of length n from a Pareto distribution.

12.4.2 networkx.utils.scipy_powerlaw_sequence

networkx.utils.**scipy_powerlaw_sequence**(*n*, *exponent=2.0*)
Return sample sequence of length n from a power law distribution.

12.4.3 networkx.utils.scipy_poisson_sequence

networkx.utils.**scipy_poisson_sequence**(*n*, *mu=1.0*)
Return sample sequence of length n from a Poisson distribution.

12.4.4 networkx.utils.scipy_uniform_sequence

networkx.utils.**scipy_uniform_sequence**(*n*)
Return sample sequence of length n from a uniform distribution.

12.4.5 networkx.utils.scipy_discrete_sequence

networkx.utils.**scipy_discrete_sequence**(*n*, *distribution=False*)

Return sample sequence of length n from a given discrete distribution.

distribution=histogram of values, will be normalized

CHAPTER

THIRTEEN

LICENSE

NetworkX is distributed with the BSD license.

CHAPTER

FOURTEEN

CITING

To cite NetworkX please use the following publication:

Aric A. Hagberg, Daniel A. Schult and Pieter J. Swart, "Exploring network structure, dynamics, and function using NetworkX", in Proceedings of the 7th Python in Science Conference (SciPy2008), Gäel Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11–15, Aug 2008

CHAPTER

FIFTEEN

CREDITS

NetworkX was originally written by Aric Hagberg, Dan Schult, and Pieter Swart with the help of many others.

Thanks to Guido van Rossum for the idea of using Python for implementing a graph data structure http://www.python.org/doc/essays/graphs.html

Thanks to David Eppstein for the idea of representing a graph G so that "for n in G" loops over the nodes in G and G[n] are node n's neighbors.

Thanks to everyone who has improved NetworkX by contributing code, bug reports (and fixes), documentation, and input on design, featues, and the future of NetworkX.

Thanks especially to the following contributors:

- Katy Bold contributed the Karate Club graph.
- Hernan Rozenfeld added dorogovtsev_goltsev_mendes_graph and did stress testing.
- Brendt Wohlberg added examples from the Stanford GraphBase.
- Jim Bagrow reported bugs in the search methods.
- Holly Johnsen helped fix the path based centrality measures.
- Arnar Flatberg fixed the graph laplacian routines.
- Chris Myers suggested using None as a default datatype, suggested improvements for the IO routines, added grid generator index tuple labeling and associated routines, and reported bugs.
- Joel Miller tested and improved the connected components methods fixed bugs and typos in the graph generators, and contributed the random clustered graph generator.
- Keith Briggs sorted out naming issues for random graphs and wrote dense_gnm_random_graph.
- Ignacio Rozada provided the Krapivsky-Redner graph generator.
- Phillipp Pagel helped fix eccentricity etc. for disconnected graphs.
- Sverre Sundsdal contributed bidirectional shortest path and Dijkstra routines, s-metric computation and graph generation
- Ross M. Richardson contributed the expected degree graph generator and helped test the pygraphviz interface.
- Christopher Ellison implemented the VF2 isomorphism algorithm and is a core developer.
- Eben Kenah contributed the strongly connected components and DFS functions.
- Sasha Gutfriend contributed edge betweenness algorithms.
- Udi Weinsberg helped develop intersection and difference operators.
- Matteo Dell'Amico wrote the random regular graph generator.

- Andrew Conway contributed ego_graph, eigenvector centrality, line graph and much more.
- Raf Guns wrote the GraphML writer.
- Salim Fadhley and Matteo Dell'Amico contributed the A* algorithm.
- Fabrice Desclaux contributed the Matplotlib edge labeling code.
- Arpad Horvath fixed the barabasi_albert_graph() generator.
- Minh Van Nguyen contributed the connected_watts_strogatz_graph() and documentation for the Graph and MultiGraph classes.
- Willem Ligtenberg contributed the directed scale free graph generator.
- Loïc Séguin-C. contributed the Ford-Fulkerson max flow and min cut algorithms, and ported all of NetworkX to Python3. He is a NetworkX core developer.
- Paul McGuire improved the performance of the GML data parser.
- Jesus Cerquides contributed the chordal graph algorithms.
- Ben Edwards contributed tree generating functions and the rich club coefficient algorithm.
- Jon Olav Vik contributed cycle finding algorithms.
- Hugh Brown improved the words.py example from the n^2 algorithm.
- Ben Reilly contributed the shapefile reader.
- Leo Lopes contributed the maximal independent set algorithm.
- Jordi Torrents contributed the bipartite clustering, bipartite node redundancy, square clustering, and other bipartite algorithms.
- Dheeraj M R contributed the distance-regular testing algorithm

CHAPTER

SIXTEEN

GLOSSARY

dictionary A Python dictionary maps keys to values. Also known as "hashes", or "associative arrays". See http://docs.python.org/tutorial/datastructures.html#dictionaries

ebunch An iteratable container of edge tuples like a list, iterator, or file.

edge Edges are either two-tuples of nodes (u,v) or three tuples of nodes with an edge attribute dictionary (u,v,dict).

edge attribute Edges can have arbitrary Python objects assigned as attributes by using keyword/value pairs when adding an edge assigning to the G.edge[u][v] attribute dictionary for the specified edge u-v.

hashable An object is hashable if it has a hash value which never changes during its lifetime (it needs a __hash__() method), and can be compared to other objects (it needs an __eq__() or __cmp__() method). Hashable objects which compare equal must have the same hash value.

Hashability makes an object usable as a dictionary key and a set member, because these data structures use the hash value internally.

All of Python's immutable built-in objects are hashable, while no mutable containers (such as lists or dictionaries) are. Objects which are instances of user-defined classes are hashable by default; they all compare unequal, and their hash value is their id().

Definition from http://docs.python.org/glossary.html

nbunch An nbunch is any iterable container of nodes that is not itself a node in the graph. It can be an iterable or an iterator, e.g. a list, set, graph, file, etc..

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.node[n] attribute dictionary for the specified node n.

BIBLIOGRAPHY

[R82] 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.

[R80] Scientific collaboration networks: II. Shortest paths, weighted networks, and centrality, M. E. J. Newman, Phys. Rev. E 64, 016132 (2001).

[R81] 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.

[R84] E. Estrada and J. A. Rodríguez-Velázquez, “Spectral measures of bipartivity in complex networks”, PhysRev E 72, 046105 (2005)

[R79] 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.

[R78] 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.

[R83] 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.

[R76] 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. http://www.steveborgatti.com/papers/bhaffiliations.pdf

[R77] 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. http://www.steveborgatti.com/papers/bhaffiliations.pdf

[R75] 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. http://www.steveborgatti.com/papers/bhaffiliations.pdf

[R85] Patrick Doreian, Vladimir Batagelj, and Anuska Ferligoj “Generalized Blockmodeling”,Cambridge University Press, 2004.

[R86] A Faster Algorithm for Betweenness Centrality. Ulrik Brandes, Journal of Mathematical Sociology 25(2):163-177, 2001. http://www.inf.uni-konstanz.de/algo/publications/b-fabc-01.pdf

[R87] Ulrik Brandes: On Variants of Shortest-Path Betweenness Centrality and their Generic Computation. Social Networks 30(2):136-145, 2008. http://www.inf.uni-konstanz.de/algo/publications/b-vspbc-08.pdf

[R92] A Faster Algorithm for Betweenness Centrality. Ulrik Brandes, Journal of Mathematical Sociology 25(2):163-177, 2001. http://www.inf.uni-konstanz.de/algo/publications/b-fabc-01.pdf

[R93] Ulrik Brandes: On Variants of Shortest-Path Betweenness Centrality and their Generic Computation. Social Networks 30(2):136-145, 2008. http://www.inf.uni-konstanz.de/algo/publications/b-vspbc-08.pdf

[R90] 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. http://www.inf.uni-konstanz.de/algo/publications/bf-cmbcf-05.pdf

[R91] Stephenson, K. and Zelen, M. Rethinking centrality: Methods and examples. Social Networks. Volume 11, Issue 1, March 1989, pp. 1-37 http://dx.doi.org/10.1016/0378-8733(89)90016-6

[R88] 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. http://www.inf.uni-konstanz.de/algo/publications/bf-cmbcf-05.pdf

[R89] A measure of betweenness centrality based on random walks, M. E. J. Newman, Social Networks 27, 39-54 (2005).

[R94] 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. http://www.inf.uni-konstanz.de/algo/publications/bf-cmbcf-05.pdf

[R95] A measure of betweenness centrality based on random walks, M. E. J. Newman, Social Networks 27, 39-54 (2005).

[R98] 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.

[R96] http://en.wikipedia.org/wiki/Tree_decomposition#Treewidth

[R97] 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

[R99] Bron, C. and Kerbosch, J. 1973. Algorithm 457: finding all cliques of an undirected graph. Commun. ACM 16, 9 (Sep. 1973), 575-577. http://portal.acm.org/citation.cfm?doid=362342.362367

[R100] 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 http://dx.doi.org/10.1016/j.tcs.2006.06.015

[R101] 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, http://dx.doi.org/10.1016/j.tcs.2008.05.010

[R103] 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

[R102] 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

[R104] Pedro G. Lind, Marta C. González, and Hans J. Herrmann. 2005 Cycles and clustering in bipartite networks. Physical Review E (72) 056127.

[R105] Depth-first search and linear graph algorithms, R. Tarjan SIAM Journal of Computing 1(2):146-160, (1972).

[R106] On finding the strongly connected components in a directed graph. E. Nuutila and E. Soisalon-Soinen Information Processing Letters 49(1): 9-14, (1994)..

[R107] Depth-first search and linear graph algorithms, R. Tarjan SIAM Journal of Computing 1(2):146-160, (1972).

[R108] On finding the strongly connected components in a directed graph. E. Nuutila and E. Soisalon-Soinen Information Processing Letters 49(1): 9-14, (1994)..

[R109] An O(m) Algorithm for Cores Decomposition of Networks Vladimir Batagelj and Matjaz Zaversnik, 2003. http://arxiv.org/abs/cs.DS/0310049

[R110] An O(m) Algorithm for Cores Decomposition of Networks Vladimir Batagelj and Matjaz Zaversnik, 2003. http://arxiv.org/abs/cs.DS/0310049

[R113] 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

[R112] 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

[R111] 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

[R114] Paton, K. An algorithm for finding a fundamental set of cycles of a graph. Comm. ACM 12, 9 (Sept 1969), 514-518.

[R115] Finding all the elementary circuits of a directed graph. D. B. Johnson, SIAM Journal on Computing 4, no. 1, 77-84, 1975. http://dx.doi.org/10.1137/0204007

[R116] Skiena, S. S. The Algorithm Design Manual (Springer-Verlag, 1998). http://www.amazon.com/exec/obidos/ASIN/0387948600/ref=ase_thealgorithmrepo/

[R119] Brouwer, A. E.; Cohen, A. M.; and Neumaier, A. Distance-Regular Graphs. New York: Springer-Verlag, 1989.

[R120] Weisstein, Eric W. "Distance-Regular Graph." http://mathworld.wolfram.com/Distance-RegularGraph.html

[R118] Weisstein, Eric W. "Intersection Array." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/IntersectionArray.html

[R117] Weisstein, Eric W. "Global Parameters." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/GlobalParameters.html

[R121] Fleury, "Deux problemes de geometrie de situation", Journal de mathematiques elementaires (1883), 257-261.

[R122] http://en.wikipedia.org/wiki/Eulerian_path

[R129] A. Langville and C. Meyer, "A survey of eigenvector methods of web information retrieval." http://citeseer.ist.psu.edu/713792.html

[R130] 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

[R131] A. Langville and C. Meyer, "A survey of eigenvector methods of web information retrieval." http://citeseer.ist.psu.edu/713792.html

[R132] 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

[R133] A. Langville and C. Meyer, "A survey of eigenvector methods of web information retrieval." http://citeseer.ist.psu.edu/713792.html

[R134] 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

[R123] A. Langville and C. Meyer, "A survey of eigenvector methods of web information retrieval." http://citeseer.ist.psu.edu/713792.html

[R124] 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.

[R125] A. Langville and C. Meyer, "A survey of eigenvector methods of web information retrieval." http://citeseer.ist.psu.edu/713792.html

[R126] 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.

[R127] A. Langville and C. Meyer, "A survey of eigenvector methods of web information retrieval." http://citeseer.ist.psu.edu/713792.html

[R128] Jon Kleinberg, Authoritative sources in a hyperlinked environment Journal of the ACM 46 (5): 604-632, 1999. doi:10.1145/324133.324140. http://www.cs.cornell.edu/home/kleinber/auth.pdf.

[R135] "Efficient Algorithms for Finding Maximum Matching in Graphs" by Zvi Galil, ACM Computing Surveys, 1986.

[R137] M. E. J. Newman, Mixing patterns in networks, Physical Review E, 67 026126, 2003

[R136] M. E. J. Newman, Mixing patterns in networks, Physical Review E, 67 026126, 2003

[R139] M. E. J. Newman, Mixing patterns in networks Physical Review E, 67 026126, 2003

[R138] M. E. J. Newman, Mixing patterns in networks Physical Review E, 67 026126, 2003

[R142] A. Barrat, M. Barthélemy, R. Pastor-Satorras, and A. Vespignani, "The architecture of complex weighted networks". PNAS 101 (11): 3747–3752 (2004).

[R143] A. Barrat, M. Barthélemy, R. Pastor-Satorras, and A. Vespignani, "The architecture of complex weighted networks". PNAS 101 (11): 3747–3752 (2004).

[R144] A. Barrat, M. Barthélemy, R. Pastor-Satorras, and A. Vespignani, "The architecture of complex weighted networks". PNAS 101 (11): 3747–3752 (2004).

[R140] A. Barrat, M. Barthélemy, R. Pastor-Satorras, and A. Vespignani, "The architecture of complex weighted networks". PNAS 101 (11): 3747–3752 (2004).

[R141] A. Barrat, M. Barthélemy, R. Pastor-Satorras, and A. Vespignani, "The architecture of complex weighted networks". PNAS 101 (11): 3747–3752 (2004).

[R145] A. Barrat, M. Barthélemy, R. Pastor-Satorras, and A. Vespignani, "The architecture of complex weighted networks". PNAS 101 (11): 3747–3752 (2004).

[R146] A. Barrat, M. Barthélemy, R. Pastor-Satorras, and A. Vespignani, "The architecture of complex weighted networks". PNAS 101 (11): 3747–3752 (2004).

[R147] 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. http://arxiv.org/abs/physics/0701290

[R148] R. Milo, N. Kashtan, S. Itzkovitz, M. E. J. Newman, U. Alon, "Uniform generation of random graphs with arbitrary degree sequences", 2006. http://arxiv.org/abs/cond-mat/0312028

[R178] Vladimir Batagelj and Ulrik Brandes, "Efficient generation of large random networks", Phys. Rev. E, 71, 036113, 2005.

[R179] 16. Erdős and A. Rényi, On Random Graphs, Publ. Math. 6, 290 (1959).

[R180] 5. (a) Gilbert, Random Graphs, Ann. Math. Stat., 30, 1141 (1959).

[R175] Donald E. Knuth, The Art of Computer Programming, Volume 2/Seminumerical algorithms, Third Edition, Addison-Wesley, 1997.

[R176] 16. Erdős and A. Rényi, On Random Graphs, Publ. Math. 6, 290 (1959).

[R177] 5. (a) Gilbert, Random Graphs, Ann. Math. Stat., 30, 1141 (1959).

[R173] 16. Erdős and A. Rényi, On Random Graphs, Publ. Math. 6, 290 (1959).

[R174] 5. (a) Gilbert, Random Graphs, Ann. Math. Stat., 30, 1141 (1959).

[R181] M. E. J. Newman and D. J. Watts, Renormalization group analysis of the small-world network model, Physics Letters A, 263, 341, 1999. http://dx.doi.org/10.1016/S0375-9601(99)00757-4

[R185] Duncan J. Watts and Steven H. Strogatz, Collective dynamics of small-world networks, Nature, 393, pp. 440–442, 1998.

[R183] 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

[R184] 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

[R172] A. L. Barabási and R. Albert "Emergence of scaling in random networks", Science 286, pp 509-512, 1999.

[R182] P. Holme and B. J. Kim, "Growing scale-free networks with tunable clustering", Phys. Rev. E, 65, 026107, 2002.

[R151] M.E.J. Newman, "The structure and function of complex networks", SIAM REVIEW 45-2, pp 167-256, 2003.

[R153] 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)

[R154] Fan Chung and L. Lu, Connected components in random graphs with given expected degree sequences, Ann. Combinatorics, 6, pp. 125-145, 2002.

[R155] 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.

[R156] G. Chartrand and L. Lesniak, "Graphs and Digraphs", Chapman and Hall/CRC, 1996.

[R152] 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

[R157] J. C. Miller "Percolation and Epidemics on Random Clustered Graphs." Physical Review E, Rapid Communication (to appear).

[R158] M.E.J. Newman, "Random clustered networks". Physical Review Letters (to appear).

[R159] P. L. Krapivsky and S. Redner, Organization of Growing Random Networks, Phys. Rev. E, 63, 066123, 2001.

[R161] P. L. Krapivsky and S. Redner, Organization of Growing Random Networks, Phys. Rev. E, 63, 066123, 2001.

[R160] P. L. Krapivsky and S. Redner, Network Growth by Copying, Phys. Rev. E, 71, 036118, 2005k.},

[R162] B. Bollob{'a}s, 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.

[R166] Penrose, Mathew, Random Geometric Graphs, Oxford Studies in Probability, 5, 2003.

[R163] Masuda, N., Miwa, H., Konno, N.: Geographical threshold graphs with small-world and scale-free properties. Physical Review E 71, 036108 (2005)

[R164] 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

[R167] B. M. Waxman, Routing of multipoint connections. IEEE J. Select. Areas Commun. 6(9),(1988) 1617-1622.

[R165] J. Kleinberg. The small-world phenomenon: An algorithmic perspective. Proc. 32nd ACM Symposium on Theory of Computing, 2000.

[R149] Jean-Loup Guillaume and Matthieu Latapy, Bipartite structure of all complex networks, Inf. Process. Lett. 90, 2004, pg. 215-221 http://dx.doi.org/10.1016/j.ipl.2004.03.007

[R150] Vladimir Batagelj and Ulrik Brandes, "Efficient generation of large random networks", Phys. Rev. E, 71, 036113, 2005.

[R170] K.B. Singer-Cohen, Random Intersection Graphs, 1995, PhD thesis, Johns Hopkins University

[R171] 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.

[R169] Godehardt, E., and Jaworski, J. Two models of random intersection graphs and their applications. Electronic Notes in Discrete Mathematics 10 (2001), 129–132.

[R168] 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.

[R188] Zachary W. An information flow model for conflict and fission in small groups. Journal of Anthropological Research, 33, 452-473, (1977).

[R189] Data file from: http://vlado.fmf.uni-lj.si/pub/networks/data/Ucinet/UciData.htm

[R186] A. Davis, Gardner, B. B., Gardner, M. R., 1941. Deep South. University of Chicago Press, Chicago, IL.

[R187] 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

[R190] Fan Chung-Graham, Spectral Graph Theory, CBMS Regional Conference Series in Mathematics, Number 92, 1997.

[R191] GEXF graph format, http://gexf.net/format/

[R192] GEXF graph format, http://gexf.net/format/

[R193] http://docs.python.org/library/pickle.html

[R194] http://docs.python.org/library/pickle.html

[R196] http://www.algorithmic-solutions.info/leda_guide/graphs/leda_native_graph_fileformat.html

[R195] http://www.algorithmic-solutions.info/leda_guide/graphs/leda_native_graph_fileformat.html

[R198] http://www.yaml.org

[R199] http://www.yaml.org

[R197] http://en.wikipedia.org/wiki/Shapefile

PYTHON MODULE INDEX

a

c

d

e

g

l

r

u

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

W