NetworkX Documentation

Release 1.0.dev1408

Aric Hagberg, Dan Schult, Pieter Swart

August 17, 2009

CONTENTS

Bibliography   271

Module Index   273

Index   275

INSTALLING

1.1 Quick Install

Get NetworkX from the Python Package Index at http://pypi.python.org/pypi/networkx

or install it with:

```
easy_install networkx
```

and an attempt will be made to find and install an appropriate version that matches your operating system and Python version.

More download options are at http://networkx.lanl.gov/download.html

1.2 Installing from Source

You can install from source by downloading a source archive file (tar.gz or zip) or by checking out the source files from the Subversion repository.

NetworkX is a pure Python package; you don't need a compiler to build or install it.

1.2.1 Source Archive File

1. Download the source (tar.gz or zip file).

2. Unpack and change directory to networkx-"version"

3. Run "python setup.py install" to build and install

4. (optional) Run "python setup_egg.py nosetests" to execute the tests

1.2.2 SVN Repository

1. Check out the networkx trunk:

```
svn co https://networkx.lanl.gov/svn/networkx/trunk networkx
```

2. Change directory to "networkx"

3. Run "python setup.py install" to build and install

4. (optional) Run "python setup_egg.py nosetests" to execute the tests

If you don't have permission to install software on your system, you can install into another directory using the –prefix or –home flags to setup.py.

For example

```
python setup.py install --prefix=/home/username/python
or
python setup.py install --home=~
```

If you didn't install in the standard Python site-packages directory you will need to set your PYTHONPATH variable to the alternate location. See http://docs.python.org/inst/search-path.html for further details.

1.3 Installing Pre-built Packages

1.3.1 Windows

Download and run the latest version of the Windows installer (.exe extension).

1.3.2 OSX 10.5

Download and install the latest mpkg.

1.3.3 Linux

Debian packages are available at http://packages.debian.org/python-networkx

1.4 Requirements

1.4.1 Python

To use NetworkX you need Python version 2.4 or later http://www.python.org/

The easiest way to get Python and most optional packages is to install the Enthought Python distribution http://www.enthought.com/products/epd.php

Other options are

Windows

- Official Python site version: http://www.python.org/download/
- ActiveState version: http://activestate.com/Products/ActivePython/

OSX

OSX 10.5 ships with Python version 2.5. If you have an older version we encourage you to download a newer release. Pre-built Python packages are available from

- Official Python site version http://www.python.org/download/
- Pythonmac http://www.pythonmac.org/packages/
- ActiveState http://activestate.com/Products/ActivePython/

If you are using Fink or MacPorts, Python is available through both of those package systems.

Linux

Python is included in all major Linux distributions

1.5 Optional packages

NetworkX will work without the following optional packages.

1.5.1 NumPy

- Download: http://scipy.org/download

1.5.2 SciPy

Provides sparse matrix representation of graphs and many numerical scientific tools.

1.5.3 Matplotlib

Provides flexible drawing of graphs

- Download: http://matplotlib.sourceforge.net/

1.5.4 GraphViz

In conjunction with either

- pygraphviz: http://networkx.lanl.gov/pygraphviz/

or

- pydot: http://dkbza.org/pydot.html

provides graph drawing and graph layout algorithms.

- Download: http://graphviz.org/

1.5.5 Other Packages

These are extra packages you may consider using with NetworkX

- IPython, interactive Python shell, http://ipython.scipy.org/
- sAsync, persistent storage with SQL, http://foss.eepatents.com/sAsync
- PyYAML, structured output format, http://pyyaml.org/

TUTORIAL

2.1 About NetworkX

NetworkX is a Python-based package for the creation, manipulation, and study of the structure, dynamics, and function of complex networks. The name means **Network "X"** and we pronounce it **NX**.

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.

The potential audience for NetworkX include: mathematicians, physicists, biologists, computer scientists, social scientists. The current state of the art of the (young and rapidly growing) 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 modern survey of Brandes and Erlebach [BE05].

Why Python? Past experience showed this approach to maximize productivity, power, multi-disciplinary scope (our application test beds included 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].

NetworkX is free software; you can redistribute it and/or modify it under the terms of the **LGPL** (GNU Lesser General Public License) as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version. Please see the license for more information.

2.1.1 NetworkX Basics

We assume you can start an interactive Python session.. We will assume that you are familiar with Python terminology (see the official Python website http://www.python.org for more information). If you did not install NetworkX into the Python site directory it might be useful to add that directory to your PYTHONPATH.

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

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

To save repetition, in the documentation we assume that NX 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. (A subclass of Graph.)

MultiDiGraph A directed version of a MultiGraph. (A subclass of DiGraph.)

Empty graph-like objects are created with

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

When called with no arguments you get a graph without any nodes or edges (empty graph). In NX every graph or network is a Python "object", and in Python the functions associated with an "object" are known as methods.

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

All graph classes have boolean attributes to describe the nature of the graph: directed, weighted, multigraph. The weighted attribute means that the edge weights are numerical, though that is not enforced. Some functions will not work on graphs that do not have weighted==True (the default), so it can be used to protect yourself against using a routine that requires numerical edge data.

The graph classes data structures are based on an adjacency list and 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 object (default 1) associated with that edge (or a list of edge objects for MultiGraph/MultiDiGraph). 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.

2.1.2 Glossary

The following shorthand is used throughout NetworkX documentation and code:

G,G1,G2,H,etc Graphs

n,n1,n2,u,v,v1,v2: Nodes (vertices)

nlist,vlist: A list of nodes (vertices)

nbunch, vbunch: A "bunch" of nodes (vertices). 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..)

e=(n1,n2), (n1,n2,x): An edge (a Python 2-tuple or 3-tuple), also written n1-n2 (if undirected) and n1->n2 (if directed).

e=(n1,n2,x): The edge object x (or list of objects for multigraphs) associated with an edge can be obtained using G.get_edge_data(n1,n2). The default G.add_edge(n1,n2) is equivalent to G.add_edge(n1,n2,1). In the case of multiple edges in multigraphs between nodes n1 and n2, one can use G.remove_edge(n1,n2) to remove all edges between n1 and n2, or G.remove_edge(n1,n2,x) to remove one edge associated with object x.

elist: A list of edges (as 2- or 3-tuples)

ebunch: A bunch of edges (as tuples). An ebunch is any iterable (non-string) container of edge-tuples. (Similar to nbunch, also see add_edge).

iterator method names: In many cases it is more efficient to iterate through items rather than creating a list of items. NetworkX provides separate methods that return an iterator. For example, G.degree() and G.edges() return lists while G.degree_iter() and G.edges_iter() return iterators.

Some potential pitfalls to be aware of:

- Although any hashable object can be used as a node, one should not change the object after it has been added as a node (since the hash can depend on the object contents).

- The ordering of objects within an arbitrary nbunch/ebunch can be machine- or implementation-dependent.

- Algorithms applicable to arbitrary nbunch/ebunch should treat them as once-through-and-exhausted iterable containers.

- len(nbunch) and len(ebunch) need not be defined.

2.1.3 Graph methods

A Graph object G has the following primitive methods associated with it. (You can use dir(G) to inspect the methods associated with object G.)

1. Non-mutating Graph methods:

```
- len(G), G.number_of_nodes(), G.order()  # number of nodes in G
- n in G,     G.has_node(n)
- for n in G:   # loop through the nodes in G
- for nbr in G[n]:  # loop through the neighbors of n in G
- G.nodes()        # list of nodes
- G.nodes_iter()   # iterator over nodes
- nbr in G[n],  G.has_edge(n1,n2)
- G.edges(), G.edges(n), G.edges(nbunch)
- G.edges_iter(), G.edges_iter(n), G.edges_iter(nbunch)
- G.get_edge_data(n1,n2)  # the object associated with edge
- G[n1][n1]  # object associated with edge, faster than get_edge_data()
- G.neighbors(n)     # list of neighbors of n
- G.neighbors_iter(n) # iterator over neighbors
- G[n]              # dictionary of neighbors of n keyed to edge object
- G.adjacency_list  #list of
- G.number_of_edges(), G.size()
- G.degree(), G.degree(n), G.degree(nbunch)
- G.degree_iter(), G.degree_iter(n), G.degree_iter(nbunch)
- G.nodes_with_selfloops()
- G.selfloop_edges()
- G.number_of_selfloops()
- G.nbunch_iter(nbunch)  # iterator over nodes in both nbunch and G

The following return a new graph::

- G.subgraph(nbunch,copy=True)
- G.copy()
- G.to_directed()
- G.to_undirected()
```

2. Mutating Graph methods:

```
    – G.add_node(n), G.add_nodes_from(nbunch)
    – G.remove_node(n), G.remove_nodes_from(nbunch)
    – G.add_edge(n1,n2), G.add_edge(*e)
    – G.add_edges_from(ebunch)
    – G.remove_edge(n1,n2), G.remove_edge(*e),
    – G.remove_edges_from(ebunch)
    – G.add_star(nlist)
    – G.add_path(nlist)
    – G.add_cycle(nlist)
    – G.clear()
    – G.subgraph(nbunch,copy=False)
```

Names of classes/objects use the CapWords convention, e.g. Graph, MultiDiGraph. Names of functions and methods use the lowercase_words_separated_by_underscores convention, e.g. petersen_graph(), G.add_node(10).

G can be inspected interactively by typing "G" (without the quotes). This will reply something like <networkx.base.Graph object at 0x40179a0c>. (On Linux machines with CPython the hexadecimal address is the memory location of the object.)

2.1.4 What Next

- A Brief Tour
- *Reference*
- Examples

2.2 A Brief Tour

Create an empty graph with no nodes and no edges.

```
>>> import networkx as nx
>>> G=nx.Graph()
```

By definition, a Graph is a collection of nodes (vertices) along with identified pairs of nodes (called edges, links, etc). In NetworkX, nodes can be any hashable object e.g. a text string, an image, an XML object, another Graph, a customized node object, etc. (Note: Python's None object should not be used as a node as it determines whether optional function arguments have been assigned in many functions.)

2.2.1 Nodes

The graph G can be grown in several ways. NetworkX includes many graph generator functions and facilities to read and write graphs in many formats. To get started though we'll look at simple manipulations. You can add one node at a time,

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

add a list of nodes,

```
>>> G.add_nodes_from([2,3])
```

or add any *nbunch* of nodes. An *nbunch* is any iterable container of nodes that is not itself a node in the graph. (e.g. a list, set, graph, file, etc..)

```
>>> H=nx.path_graph(10)
>>> G.add_nodes_from(H)
```

Note that G now contains the nodes of H as nodes of G. In contrast, you could use the graph H as a node in G.

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

The graph G now contains H as a node. This flexibility is very powerful as it allows graphs of graphs, graphs of files, graphs of functions and much more. It is worth thinking about how to structure your application so that the nodes are useful entities. Of course you can always use a unique identifier in G and have a separate dictionary keyed by identifier to the node information if you prefer. (Note: You should not change the node object if the hash depends on its contents.)

2.2.2 Edges

G can also be grown by adding one edge at a time,

```
>>> G.add_edge(1,2)
>>> e=(2,3)
>>> G.add_edge(*e) # unpack edge tuple*
```

by adding a list of edges,

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

or by adding any *ebunch* of edges, An *ebunch* is any iterable container of edge-tuples. An edge-tuple can be a 2-tuple of nodes or a 3-tuple with 2 nodes followed by an edge attribute dictionary, e.g. (2,3,{'weight':3.1415}). Edge attributes are discussed further below

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

One can demolish the graph in a similar fashion; using remove_node, remove_nodes_from, remove_edge and remove_edges_from, e.g.

```
>>> G.remove_node(H)
```

There are no complaints when adding existing nodes or edges. For example, after removing all nodes and edges,

```
>>> G.clear()
```

we add new nodes/edges and NetworkX quietly ignores any that are already present.

```
>>> G.add_edges_from([(1,2),(1,3)])
>>> G.add_node(1)
>>> G.add_edge(1,2)
>>> G.add_node("spam")        # adds node "spam"
>>> G.add_nodes_from("spam") # adds 4 nodes: 's', 'p', 'a', 'm'
```

At this stage the graph G consists of 8 nodes and 2 edges, as can be seen by:

```
>>> G.number_of_nodes()
8
>>> G.number_of_edges()
2
```

We can examine them with

```
>>> G.nodes()
['a', 1, 2, 3, 'spam', 'm', 'p', 's']
>>> G.edges()
[(1, 2), (1, 3)]
>>> G.neighbors(1)
[2, 3]
```

Removing nodes or edges has similar syntax to adding:

```
>>> G.remove_nodes_from("spam")
>>> G.nodes()
[1, 2, 3, 'spam']
>>> G.remove_edge(1,3)
```

You can specify graph data upon instantiation if an appropriate structure exists.

```
>>> H=nx.DiGraph(G)   # create a DiGraph using the connections from G
>>> H.edges()
[(1, 2), (2, 1)]
>>> H=nx.Graph({0:[1,2,3], 1:[0,3], 2:[0], 3:[0]})   # dict-of-lists adjacency
```

2.2.3 Edge Objects

Edge data/weights/labels/objects can also be associated with an edge. Each edge has an attribute dictionary associated with it. Arbitrary key=value attributes can be assigned. The special attribute 'weight' should be numeric and holds values used by algorithms requiring weighted edges.

```
>>> H=nx.Graph()
>>> H.add_edge(1,2,color='red')
>>> H.add_edges_from([(1,3,{'color':'blue'}), (2,0,{'color':'red'}), (0,3)])
>>> H.edges()
[(0, 2), (0, 3), (1, 2), (1, 3)]
>>> H.edges(data=True)
[(0, 2, {'color': 'red'}), (0, 3, {}), (1, 2, {'color': 'red'}), (1, 3, {'color': 'blue'})]
```

To update the edge attributes for an existing edge, add the edge again with the new value. (Note: with MultiGraph you need to keep track of the edge key for the edge you want to update.)

```
>>> H.add_edge(0,2,color='blue')
>>> H.edges(data=True)
[(0, 2, {'color': 'blue'}), (0, 3, {}), (1, 2, {'color': 'red'}), (1, 3, {'color': 'blue'})]
```

You might notice that nodes and edges are not NetworkX objects. This leaves you free to use your existing node and edge objects, or more typically, use numerical values or strings where appropriate. A node can be any hashable object (except None), and an edge can be associated with any object x using G.add_edge(n1,n2,object=x).

As an example, n1 and n2 could be protein objects from the RCSB Protein Data Bank, and x could refer to an XML record of publications detailing experimental observations of their interaction.

We have found this power quite useful, but its abuse can lead to unexpected surprises unless one is familiar with Python. If in doubt, consider using nx.convert_node_labels_to_integers() to obtain a more traditional graph with integer labels.

2.2.4 Accessing Edges/Neighbors

In addition to the methods nodes(), edges(), and neighbors(), iterator versions (e.g. edges_iter()) can save you from creating large lists when you are just going to iterate through them anyway.

Fast direct access to the graph data structure is also possible using subscript notation. Warning: do not change the returned dict–it is part of the graph data structure and direct manipulation may leave the graph in an inconsistent state.

```
>>> G[1]    # Warning: do not change the resulting dict
{2: {}}
>>> G[1][2]
{}
```

You can safely set the attributes of an edge using subscript notation if the edge aleady exists.

```
>>> G.add_edge(1,3)
>>> G[1][3]['color']='blue'
```

Fast examination of all edges is achieved using adjacency iterators. Note that for undirected graphs this actually looks at each edge twice.

```
>>> FG=nx.Graph()
>>> FG.add_weighted_edges_from([(1,2,0.125),(1,3,0.75),(2,4,1.2),(3,4,0.375)])
>>> for n,nbrs in FG.adjacency_iter():
...     for nbr,eattr in nbrs.iteritems():
...         data=eattr['weight']
...         if data<0.5: print (n,nbr,data)
(1, 2, 0.125)
(2, 1, 0.125)
(3, 4, 0.375)
(4, 3, 0.375)
```

2.2.5 Directed Graphs

The DiGraph class provides additional methods specific to directed edges, e.g. G.out_edges(), G.in_degree(), G.predecessors(), G.successors() etc. To allow algorithms to work with both classes easily, the directed versions of neighbors() and degree() are equivalent to successors() and the sum of in_degree() and out_degree() respectively even though that may feel inconsistent at times.

```
>>> DG=nx.DiGraph()
>>> DG.add_weighted_edges_from([(1,2,0.5), (3,1,0.75)])
>>> DG.out_degree(1,weighted=True)
0.5
>>> DG.degree(1,weighted=True)
1.25
>>> DG.successors(1)
[2]
>>> DG.neighbors(1)
[2]
```

Some algorithms work only for directed graphs and others are not well defined for directed graphs. Indeed the tendency to lump directed and undirected graphs together is dangerous. If you want to treat a directed graph as undirected for some measurement you should probably convert it using G.to_undirected() or nx.Graph(G).

2.2.6 MultiGraph and MultiDiGraph

NetworkX provides classes for graphs which allow multiple edges between any pair of nodes. The MultiGraph and MultiDiGraph classes allow you to add the same edge twice, possibly with different edge data. This can be powerful for some applications, but many algorithms are not well defined on such graphs. Shortest path is one example. Where results are well defined, e.g. degree() we provide the function. Otherwise you should convert to a standard graph in a way that makes the measurement well defined.

```
>>> MG=nx.MultiGraph()
>>> MG.add_weighted_edges_from([(1,2,.5), (1,2,.75), (2,3,.5)])
>>> MG.degree(weighted=True, with_labels=True)
{1: 1.25, 2: 1.75, 3: 0.5}
>>> GG=nx.Graph()
>>> for n,nbrs in MG.adjacency_iter():
...     for nbr,edict in nbrs.iteritems():
...         minvalue=min(edict.values())
...         GG.add_edge(n,nbr,minvalue)


>>> nx.shortest_path(GG,1,3)
[1, 2, 3]
```

2.2.7 Graph generators and graph operations

In addition to constructing graphs node-by-node or edge-by-edge, they can also be generated by

1. Applying classic graph operations, such as:

```
subgraph(G, nbunch)        - induce subgraph of G on nodes in nbunch
union(G1,G2)               - graph union
disjoint_union(G1,G2)      - graph union assuming all nodes are different
cartesian_product(G1,G2)   - return Cartesian product graph
compose(G1,G2)             - combine graphs identifying nodes common to both
complement(G)              - graph complement
create_empty_copy(G)       - return an empty copy of the same graph class
convert_to_undirected(G)   - return an undirected representation of G
convert_to_directed(G)     - return a directed representation of G
```

2. Using a call to one of the classic small graphs, e.g.

```
>>> petersen=nx.petersen_graph()
>>> tutte=nx.tutte_graph()
>>> maze=nx.sedgewick_maze_graph()
>>> tet=nx.tetrahedral_graph()
```

1. Using a (constructive) generator for a classic graph, e.g.

```
>>> K_5=nx.complete_graph(5)
>>> K_3_5=nx.complete_bipartite_graph(3,5)
>>> barbell=nx.barbell_graph(10,10)
>>> lollipop=nx.lollipop_graph(10,20)
```

1. Using a stochastic graph generator, e.g.

```
>>> er=nx.erdos_renyi_graph(100,0.15)
>>> ws=nx.watts_strogatz_graph(30,3,0.1)
>>> ba=nx.barabasi_albert_graph(100,5)
>>> red=nx.random_lobster(100,0.9,0.9)
```

1. Reading a graph stored in a file using common graph formats, such as edge lists, adjacency lists, GML, GraphML, pickle, LEDA and others.

```
>>> mygraph=nx.read_graphml("path/to/file")
>>> fh=open("path/to/file")
>>> mygraph=nx.read_graphml(fh)
```

Details on graph formats: *Reading and Writing*

Details on graph generator functions: *Graph generators*

2.2.8 Analyzing graphs

The structure of G can be analyzed using various graph-theoretic functions such as:

```
>>> nx.connected_components(G)
[[1, 2, 3], ['spam']]
```

```
>>> sorted(nx.degree(G))
[0, 1, 1, 2]
```

```
>>> nx.clustering(G)
[0.0, 0.0, 0.0, 0.0]
```

With no nodes specified, functions that return Node Properties will return a list of values in an arbitrary order determined by the internal python dictionary structure of the graph (which is returned by G.nodes() though it can change if the dictionary is resized).

The keyword argument with_labels=True returns a dict keyed by nodes to the node values.

```
>>> nx.degree(G, with_labels=True)
{1: 2, 2: 1, 3: 1, 'spam': 0}
```

Functions that return Node Properties, e.g. degree(), clustering(), etc, can For values of specific nodes, you can provide a single node or an nbunch of nodes as argument. If a single node is specified, then a single value is returned. If an nbunch is specified, then the function will return a list of values.

```
>>> nx.degree(G,1)
2
>>> G.degree(1)
2
>>> sorted(G.degree([1,2]))
[1, 2]
>>> sorted(G.degree())
[0, 1, 1, 2]
>>> G.degree([1,2],with_labels=True)
{1: 2, 2: 1}
```

Details on graph algorithms supported: *Algorithms*

2.2.9 Drawing Graphs

NetworkX is not primarily a graph drawing package but basic drawing with Matplotlib as well as an interface to use the open source Graphviz software package are included. These are part of the networkx.drawing package and will be imported if possible. See *Drawing* for details.

First import Matplotlib's plot interface (pylab works too)

```
>>> import matplotlib.pyplot as plt
```

You may find it useful to interactively test code using "ipython -pylab", which combines the power of ipython and matplotlib and provides a convenient interactive mode.

To test if the import of networkx.drawing was successful draw G using one of

```
>>> nx.draw(G)
>>> nx.draw_random(G)
>>> nx.draw_circular(G)
>>> nx.draw_spectral(G)
```

when drawing to an interactive display. Note that you may need to issue a Matplotlib

```
>>> plt.show()
```

command if you are not using matplotlib in interactive mode: (See Matplotlib FAQ)

To save drawings to a file, use, for example

```
>>> nx.draw(G)
>>> plt.savefig("path.png")
```

writes to the file "path.png" in the local directory. If Graphviz and PyGraphviz, or pydot, are available on your system, you can also use

```
>>> nx.draw_graphviz(G)
>>> nx.write_dot(G,'file.dot')
```

Details on drawing graphs: *Drawing*

2.2.10 What Next

Now that you have an idea of what the NetworkX package provides, you should investigate the parts of the package most useful for you.

Reference Section provides details on NetworkX.

/examples/index provides some example programs written using NetworkX.

2.3 References

REFERENCE

Release 1.0.dev1408

Date August 17, 2009

3.1 Introduction

NetworkX provides data structures for graphs (or networks) along with graph algorithms, generators, and drawing tools.

3.1.1 History and 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.

History:

- NetworkX was inspired by Guido van Rossum's 1998 Python graph representation essay:ref:*[1]<references>*.

- First public release in April 2005. Version 1.0 released in 2009.

3.1.2 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).

3.1.3 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. Our 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.

- multiedges: 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*

A third graph attribute (**weighted**) signifies whether the edge data is numeric. Some algorithms depend on numeric edge data. By default this attribute is True, but if you use more complicated edge data you may want to set it to False so that any algorithms that using numeric edge data can warn you if you try to use them.

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.

Edges often have data associated with them. If the data is numeric, the graph is **weighted**. Some of the graph algorithms, such as Dijkstra's shortest path algorithm, will not work unless the edge data is a number. In other cases more than one attribute is associated with an edge. Here you have to make a choice based on how you plan to use your data. If you will mostly examine each attribute over the entire graph, it may be best to create more than one graph object–each corresponding to a given attribute. If you will be traversing the edges of the network and looking at more than one attribute of each edge, then it may make more sense to create a single graph object with more complicated data representing each edge. Edge data is arbitrary in NetworkX so you can make it anything from a tuple to a customized object for your application.

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.

3.1.4 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.

Manual addition and removing 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)  # you can specify edge data
>>> import math
>>> G.add_edge('y','x',function=math.cos) # edge attributes can be anything
>>> G.add_node(math.cos) # any hashable can be a node


>>> elist=[('a','b',0.3),('b','c',0.9),('a','c',0.5),('c','d',1.2)]
>>> G.add_weighted_edges_from(elist)   # add multiple edges at once
```

You can see the *Tutorial* for more examples. Some basic graph operations such as union and intersection are described in the *operators module* documentation.

Graph generators such as binomial_graph and powerlaw_graph are provided in the `generators` subpackage.

For importing network data from formats such as GML, GraphML, edge list text files see the `readwrite` subpackage.

3.1.5 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*.

3.1.6 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 Google group or the developer website.

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']
```

3.1.7 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 dict 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.

3.1.8 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 dict keyed by neighbor to the edge attribute dict. 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 where the third dict is keyed by an edge key identifier to the fourth dict 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 dict 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
```

The dict-of-dicts data structure is based on the following

References

Guido van Rossum. Python Patterns - Implementing Graphs, 1998. http://www.python.org/doc/essays/graphs/

David Eppstein. {PADS}, a library of {P}ython {A}lgorithms and {D}ata {S}tructures, 2008.

3.2 Graph types

NetworkX provides data structures and methods for storing graphs.

All NetworkX graph classes allow (hashable) Python objects as nodes. Any Python object can be assigned to edges. The choice of graph class depends on the structure of the graph you want to represent.

3.2.1 Basic graph types

Graph – Undirected graphs with self loops

Overview

Graph (*data=None, name=", **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.
>
> **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:**

> DiGraph, MultiGraph, MultiDiGraph

Examples

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

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

G can be grown in several ways.

Nodes:

Add one node at a time:

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

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

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

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

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

Edges:

G can also be grown by adding edges.

Add one edge,

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

a list of edges,

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

or a collection of edges,

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

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

Attributes:

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

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

Add node attributes using add_node(), add_nodes_from() or G.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
>>> print [n for n in G if n<3]    # iterate through nodes
[1, 2]
>>> print len(G)   # number of nodes in graph
5
>>> print 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.iteritems():
...         if 'weight' in eattr:
...             print (n,nbr,eattr['weight'])
(1, 2, 4)
(2, 1, 4)
(2, 3, 8)
(3, 2, 8)
>>> print [ (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.

Methods

Adding and Removing Nodes and Edges	Graph.__init__(**attr[, data, name])	Initialize a graph with edges, name,
	Graph.add_node(n, **attr[, attr_dict])	Add a single node n and update nod
	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 weig specified weights.
	Graph.remove_edge(u, v)	Remove the edge between u and v.
	Graph.remove_edges_from(ebunch)	Remove all edges specified in ebun
	Graph.add_star(nlist, **attr)	Add a star.
	Graph.add_path(nlist, **attr)	Add a path.
	Graph.add_cycle(nlist, **attr)	Add a cycle.
	Graph.clear()	Remove all nodes and edges from t

networkx.Graph.__init__

__init__ (*data=None, name='', **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

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

add_nodes_from(*nodes, **attr*)

>   Add multiple nodes.

>   **Parameters nodes** : iterable container

>   A container of nodes (list, dict, set, etc.). The container will be iterated through once.

>   **attr** : keyword arguments, optional (default= no attributes)

>   Update attributes for all nodes in nodes.

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())
[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)
```

networkx.Graph.remove_node

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

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

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

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

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

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

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

add_star (*nlist, **attr*)

> Add a star.

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

> > **Parameters nlist** : list
> >
> > > A list 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

add_path(*nlist, **attr*)

Add a path.

> **Parameters nlist** : list
>
> > A list 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

add_cycle(*nlist, **attr*)

Add a cycle.

> **Parameters nlist** : list
>
> > A list 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

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()
[]
```

	`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 (
Iterating over nodes and edges	`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 fo
	`Graph.nbunch_iter([nbunch])`	Return an iterator of nodes contained in nbunch that

networkx.Graph.nodes

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

networkx.Graph.nodes_iter

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'.

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

Examples

```python
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> for n in G.nodes_iter():
...     print n,
0 1 2
>>> for n,d in G.nodes_iter(data=True):
...     print d,
{} {} {}
```

networkx.Graph.__iter__

__iter__ ()

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

> **Returns niter** : iterator
>
> > An iterator over all nodes in the graph.

Examples

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

networkx.Graph.edges

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

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

```pycon
>>> 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

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].

```pycon
>>> 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,

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

Examples

```pycon
>>> 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

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

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. >>> for n in G[0]: ... print n 1

Examples

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

networkx.Graph.__getitem__

__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])
>>> print G[0]
{1: {}}
```

networkx.Graph.adjacency_list

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

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
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.

	`Graph.has_node`(n)	Return True if the graph contains the no
	`Graph.__contains__`(n)	Return True if n is a node, False otherwi
	`Graph.has_edge`(u, v)	Return True if the edge (u,v) is in the gr
	`Graph.nodes_with_selfloops`()	Return a list of nodes with self loops.
	`Graph.selfloop_edges`([data])	Return a list of selfloop edges.
	`Graph.order`()	Return the number of nodes in the graph
Information about graph structure	`Graph.number_of_nodes`()	Return the number of nodes in the graph
	`Graph.__len__`()	Return the number of nodes.
	`Graph.size`([weighted])	Return the number of edges.
	`Graph.number_of_edges`([u, v])	Return the number of edges between tw
	`Graph.number_of_selfloops`()	Return the number of selfloop edges.
	`Graph.degree`([nbunch, with_labels, weighted])	Return the degree of a node or nodes.
	`Graph.degree_iter`([nbunch, weighted])	Return an iterator for (node, degree).

networkx.Graph.has_node

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])
>>> print G.has_node(0)
True
```

It is more readable and simpler to use

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

networkx.Graph.__contains__

__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])
>>> print 1 in G
True
```

networkx.Graph.has_edge

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.nodes_with_selfloops

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

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.order

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

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])
>>> print len(G)
3
```

networkx.Graph.__len__

__**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])
>>> print len(G)
4
```

networkx.Graph.size

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
```

networkx.Graph.number_of_edges

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.number_of_selfloops

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
```

networkx.Graph.degree

degree(*nbunch=None, with_labels=False, 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.

> > **with_labels** : bool, optional (default=False)

> > > If True return a dictionary of degrees keyed by node.

weighted : bool, optional (default=False)

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

Returns nd : list, or dictionary

A list of node degrees or a dictionary with nodes as keys and degree as values if
with_labels=True).

Examples

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

networkx.Graph.degree_iter

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

```python
>>> 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)]
```

	Graph.copy()	Return a copy of the graph.
Making copies and subgraphs	Graph.to_directed()	Return a directed representation of the graph.
	Graph.subgraph(nbunch[, copy])	Return the subgraph induced on nodes in nbunch.

networkx.Graph.copy
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_directed
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 is similar to DiGraph(self) which returns a shallow copy. self.to_undirected() returns a deepcopy of edge, node and graph attributes.
>
> ### 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

subgraph (*nbunch, copy=True*)

> Return the subgraph induced on nodes in nbunch.
>
> The induced subgraph of the graph has the nodes in nbunch as its node set and the edges adjacent to those nodes as its edge set.
>
> > **Parameters nbunch** : list, iterable
> >
> > > A container of nodes. The container will be iterated through once.
> >
> > **copy** : bool, optional (default=True)
> >
> > > If True return a new graph holding the subgraph including copies of all edge and node properties. If False the subgraph is created using the original graph by deleting all nodes not in nbunch (this changes the original graph).
> >
> > **Returns G** : Graph
> >
> > > A subgraph of the graph. If copy=True a new graph is returned with copies of graph, node, and edge data. If copy=False the subgraph is created in place by modifying the original graph.

Notes

If copy=True, nodes and edges are copied using copy.deepcopy.

Examples

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

DiGraph - Directed graphs with self loops

Overview

DiGraph (*data=None, name=", **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.

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:

Graph, MultiGraph, MultiDiGraph

Examples

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

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

G can be grown in several ways.

Nodes:

Add one node at a time:

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

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

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

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

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

Edges:

G can also be grown by adding edges.

Add one edge,

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

a list of edges,

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

or a collection of edges,

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

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

Attributes:

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

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

Add node attributes using add_node(), add_nodes_from() or G.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
>>> print [n for n in G if n<3]    # iterate through nodes
[1, 2]
>>> print len(G)  # number of nodes in graph
5
>>> print 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.iteritems():
...         if 'weight' in eattr:
...             print (n,nbr,eattr['weight'])
(1, 2, 4)
(2, 3, 8)
>>> print [ (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.

Methods

	`DiGraph.__init__`(**attr[, data, name])	Initialize a graph with edges, nam
	`DiGraph.add_node`(n, **attr[, attr_dict])	Add a single node n and update no
	`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.
Adding and Removing Nodes and Edges	`DiGraph.add_weighted_edges_from`(ebunch, **attr)	Add all the edges in ebunch as we specified weights.
	`DiGraph.remove_edge`(u, v)	Remove the edge between u and v
	`DiGraph.remove_edges_from`(ebunch)	Remove all edges specified in ebu
	`DiGraph.add_star`(nlist, **attr)	Add a star.
	`DiGraph.add_path`(nlist, **attr)	Add a path.
	`DiGraph.add_cycle`(nlist, **attr)	Add a cycle.
	`DiGraph.clear`()	Remove all nodes and edges from

networkx.DiGraph.__init__

__init__ (*data=None, name='', **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

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

add_nodes_from(*nodes, **attr*)

> Add multiple nodes.
>
> > **Parameters** **nodes** : iterable container
> >
> > > A container of nodes (list, dict, set, etc.). The container will be iterated through once.
> >
> > > **attr** : keyword arguments, optional (default= no attributes)
> > >
> > > > Update attributes for all nodes in nodes.
>
> **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())
[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)
```

networkx.DiGraph.remove_node

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

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

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:

```python
>>> 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:

```python
>>> G.add_edge(1, 2, weight=3)
>>> G.add_edge(1, 3, weight=7, capacity=15, length=342.7)
```

networkx.DiGraph.add_edges_from

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

```python
>>> 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

```python
>>> 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

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

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

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

add_star (*nlist, **attr*)

Add a star.

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

> **Parameters nlist** : list
>
> > A list 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

add_path(*nlist, **attr*)

> Add a path.

> > **Parameters nlist** : list

> > > A list 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

add_cycle(*nlist, **attr*)

> Add a cycle.

> > **Parameters nlist** : list

> > > A list 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

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
	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 fo
	DiGraph.nbunch_iter([nbunch])	Return an iterator of nodes contained in nbunch that graph.

networkx.DiGraph.nodes

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])
>>> print G.nodes()
```

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

networkx.DiGraph.nodes_iter

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])
>>> for n in G:
...     print n,
0 1 2
```

Examples

```
>>> G = nx.Graph()   # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> for n in G.nodes_iter():
...     print n,
0 1 2
>>> for n,d in G.nodes_iter(data=True):
...     print d,
{} {} {}
```

networkx.DiGraph.__iter__

__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])
>>> for n in G:
...     print n,
0 1 2 3
```

networkx.DiGraph.edges

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

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

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

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

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

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

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

neighbors(*n*)

Return a list of successor nodes of n.

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

networkx.DiGraph.neighbors_iter

neighbors_iter(*n*)

Return an iterator over successor nodes of n.

neighbors_iter() and successors_iter() are the same.

networkx.DiGraph.__getitem__

__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])
>>> print G[0]
{1: {}}
```

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

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

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

> neighbors_iter() and successors_iter() are the same.

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

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

networkx.DiGraph.adjacency_list
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

`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

`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.

	`DiGraph.has_node`(n)	Return True if the graph contains the no
	`DiGraph.__contains__`(n)	Return True if n is a node, False otherwi expression
	`DiGraph.has_edge`(u, v)	Return True if the edge (u,v) is in the gr
	`DiGraph.nodes_with_selfloops`()	Return a list of nodes with self loops.
	`DiGraph.selfloop_edges`([data])	Return a list of selfloop edges.
	`DiGraph.order`()	Return the number of nodes in the graph
Information about graph structure	`DiGraph.number_of_nodes`()	Return the number of nodes in the graph
	`DiGraph.__len__`()	Return the number of nodes.
	`DiGraph.size`([weighted])	Return the number of edges.
	`DiGraph.number_of_edges`([u, v])	Return the number of edges between tw
	`DiGraph.number_of_selfloops`()	Return the number of selfloop edges.
	`DiGraph.degree`([nbunch, with_labels, weighted])	Return the degree of a node or nodes.
	`DiGraph.degree_iter`([nbunch, weighted])	Return an iterator for (node, degree).

networkx.DiGraph.has_node

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])
>>> print G.has_node(0)
True
```

It is more readable and simpler to use

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

networkx.DiGraph.__contains__

__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])
>>> print 1 in G
True
```

networkx.DiGraph.has_edge

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.nodes_with_selfloops

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

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.order

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

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])
>>> print len(G)
3
```

networkx.DiGraph.__len__

__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])
>>> print len(G)
4
```

networkx.DiGraph.size

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
```

networkx.DiGraph.number_of_edges

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.number_of_selfloops

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
```

networkx.DiGraph.degree

degree (*nbunch=None, with_labels=False, 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.

with_labels : bool, optional (default=False)

If True return a dictionary of degrees keyed by node.

weighted : bool, optional (default=False)

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

Returns **nd** : list, or dictionary

A list of node degrees or a dictionary with nodes as keys and degree as values if with_labels=True).

Examples

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

networkx.DiGraph.degree_iter

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)]
```

Making copies and subgraphs

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

networkx.DiGraph.copy

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

to_undirected()

> Return an undirected representation of the 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 is similar to Graph(self) which returns a shallow copy. self.to_undirected() returns a deepcopy of edge, node and graph attributes.

networkx.DiGraph.subgraph

subgraph(*nbunch, copy=True*)

   Return the subgraph induced on nodes in nbunch.

   The induced subgraph of the graph has the nodes in nbunch as its node set and the edges adjacent to those nodes as its edge set.

   > **Parameters nbunch** : list, iterable
   >
   > > A container of nodes. The container will be iterated through once.
   >
   > **copy** : bool, optional (default=True)
   >
   > > If True return a new graph holding the subgraph including copies of all edge and node properties. If False the subgraph is created using the original graph by deleting all nodes not in nbunch (this changes the original graph).
   >
   > **Returns G** : Graph
   >
   > > A subgraph of the graph. If copy=True a new graph is returned with copies of graph, node, and edge data. If copy=False the subgraph is created in place by modifying the original graph.

   ### Notes

   If copy=True, nodes and edges are copied using copy.deepcopy.

   ### Examples

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

networkx.DiGraph.reverse

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).

MultiGraph - Undirected graphs with self loops and parallel edges

Overview

MultiGraph(*data=None, name=", **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.
>
> **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:

`Graph`, `DiGraph`, `MultiDiGraph`

Examples

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

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

G can be grown in several ways.

Nodes:

Add one node at a time:

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

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

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

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

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

Edges:

G can also be grown by adding edges.

Add one edge,

```
>>> 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
>>> print [n for n in G if n<3]   # iterate through nodes
[1, 2]
>>> print len(G)   # number of nodes in graph
5
>>> print 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.iteritems():
...          for key,eattr in keydict.iteritems():
...              if 'weight' in eattr:
...                  print (n,nbr,eattr['weight'])
(1, 2, 4)
(2, 1, 4)
(2, 3, 8)
(3, 2, 8)
>>> print [ (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.

Methods

	`MultiGraph.__init__`(**attr[, data, name])	Initialize a graph with edges, nam
	`MultiGraph.add_node`(n, **attr[, attr_dict])	Add a single node n and update n
	`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.
Adding and Removing Nodes and Edges	`MultiGraph.add_weighted_edges_from`(ebunch, ...)	Add all the edges in ebunch as we specified weights.
	`MultiGraph.remove_edge`(u, v[, key])	Remove the edge between u and v
	`MultiGraph.remove_edges_from`(ebunch)	Remove all edges specified in ebu
	`MultiGraph.add_star`(nlist, **attr)	Add a star.
	`MultiGraph.add_path`(nlist, **attr)	Add a path.
	`MultiGraph.add_cycle`(nlist, **attr)	Add a cycle.
	`MultiGraph.clear`()	Remove all nodes and edges from

networkx.MultiGraph.__init__

__init__(*data=None, name=", **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

> ```python
> >>> 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

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

networkx.MultiGraph.add_node

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

add_nodes_from(*nodes, **attr*)

Add multiple nodes.

> **Parameters nodes** : iterable container
>
> > A container of nodes (list, dict, set, etc.). The container will be iterated through once.
>
> > **attr** : keyword arguments, optional (default= no attributes)
>
> > Update attributes for all nodes in nodes.

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())
[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)
```

networkx.MultiGraph.remove_node

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

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

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

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

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

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

    Remove the edge between u and v.

        Parameters u,v: nodes :

            Remove edge or edges between nodes u and v.

         key : hashable identifier, optional (default= None)

            Used to distinguish multiedges between a pair of nodes. If None, remove all edges between 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.MultiGraph()   # or MultiDiGraph, 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
>>> G.remove_edge(2,3,key=0) # identify an individual edge with key
```

networkx.MultiGraph.remove_edges_from

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

networkx.MultiGraph.add_star

add_star (*nlist, **attr*)

Add a star.

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

Parameters nlist : list

A list 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

add_path (*nlist, **attr*)

Add a path.

Parameters nlist : list

A list 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
add_cycle(*nlist, **attr*)
    Add a cycle.

> **Parameters nlist** : list
>
> > A list 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
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()
[]
```

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 e
MultiGraph.neighbors(n)	Return a list of the nodes connected to the node
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 gra
MultiGraph.adjacency_iter()	Return an iterator of (node, adjacency dict) tuple
MultiGraph.nbunch_iter([nbunch])	Return an iterator of nodes contained in nbunch the graph.

(Row label at left: **Iterating over nodes and edges**)

networkx.MultiGraph.nodes

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

networkx.MultiGraph.nodes_iter

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])
>>> for n in G:
...     print n,
0 1 2
```

Examples

```
>>> G = nx.Graph()   # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> for n in G.nodes_iter():
...     print n,
0 1 2
>>> for n,d in G.nodes_iter(data=True):
...     print d,
{} {} {}
```

networkx.MultiGraph.__iter__

__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])
>>> for n in G:
...     print n,
0 1 2 3
```

networkx.MultiGraph.edges

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).
>
> **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.MultiGraph.edges_iter

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.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.MultiGraph.get_edge_data

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.
> >
> > **Returns edge_dict** : dictionary
> >
> > > The edge attribute dictionary.

Notes

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

```
>>> G = nx.MultiGraph()
>>> G.add_path([0,1,2,3])
>>> G[0][1][0]
{}
```

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

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

Examples

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

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

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. >>> for n in G[0]: ... print n 1

Examples

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

networkx.MultiGraph.__getitem__

__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])
>>> print G[0]
{1: {}}
```

networkx.MultiGraph.adjacency_list

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

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

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.

	`MultiGraph.has_node`(n)	Return True if the graph contains the
	`MultiGraph.__contains__`(n)	Return True if n is a node, False other
	`MultiGraph.has_edge`(u, v[, key])	Return True if the edge (u,v) is in the
	`MultiGraph.nodes_with_selfloops`()	Return a list of nodes with self loops.
	`MultiGraph.selfloop_edges`([data])	Return a list of selfloop edges.
	`MultiGraph.order`()	Return the number of nodes in the gra
Information about graph structure	`MultiGraph.number_of_nodes`()	Return the number of nodes in the gra
	`MultiGraph.__len__`()	Return the number of nodes.
	`MultiGraph.size`([weighted])	Return the number of edges.
	`MultiGraph.number_of_edges`([u, v])	Return the number of edges between t
	`MultiGraph.number_of_selfloops`()	Return the number of selfloop edges.
	`MultiGraph.degree`([nbunch, with_labels, ...])	Return the degree of a node or nodes.
	`MultiGraph.degree_iter`([nbunch, weighted])	Return an iterator for (node, degree).

networkx.MultiGraph.has_node

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])
>>> print G.has_node(0)
True
```

It is more readable and simpler to use

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

networkx.MultiGraph.__contains__

__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])
>>> print 1 in G
True
```

networkx.MultiGraph.has_edge

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

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.MultiGraph.nodes_with_selfloops
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
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.MultiGraph.order

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

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])
>>> print len(G)
3
```

networkx.MultiGraph.__len__

__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])
>>> print len(G)
4
```

networkx.MultiGraph.size

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
```

networkx.MultiGraph.number_of_edges

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.number_of_selfloops

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
```

networkx.MultiGraph.degree

degree(*nbunch=None, with_labels=False, 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.
>>
>> **with_labels** : bool, optional (default=False)
>>
>>> If True return a dictionary of degrees keyed by node.

> **weighted** : bool, optional (default=False)
>
>> If True return the sum of edge weights adjacent to the node.
>
> **Returns nd** : list, or dictionary
>
>> A list of node degrees or a dictionary with nodes as keys and degree as values if with_labels=True).

Examples

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

networkx.MultiGraph.degree_iter

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)]
```

	MultiGraph.copy()	Return a copy of the graph.
Making copies and subgraphs	MultiGraph.to_directed()	Return a directed representation of the graph.
	MultiGraph.subgraph(nbunch[, copy])	Return the subgraph induced on nodes in nbunch.

3.2. Graph types   **95**

networkx.MultiGraph.copy

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_directed

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 is similar to MultiDiGraph(self) which returns a shallow copy. self.to_undirected() returns a deepcopy of edge, node and graph attributes.

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

subgraph (*nbunch, copy=True*)

> Return the subgraph induced on nodes in nbunch.
>
> The induced subgraph of the graph has the nodes in nbunch as its node set and the edges adjacent to those nodes as its edge set.
>
> > **Parameters nbunch** : list, iterable
> >
> > > A container of nodes. The container will be iterated through once.
> >
> > **copy** : bool, optional (default=True)
> >
> > > If True return a new graph holding the subgraph including copies of all edge and node properties. If False the subgraph is created using the original graph by deleting all nodes not in nbunch (this changes the original graph).
> >
> > **Returns G** : Graph
> >
> > > A subgraph of the graph. If copy=True a new graph is returned with copies of graph, node, and edge data. If copy=False the subgraph is created in place by modifying the original graph.

Notes

If copy=True, nodes and edges are copied using copy.deepcopy.

Examples

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

MultiDiGraph - Directed graphs with self loops and parallel edges

Overview

MultiDiGraph (*data=None, name='', **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.

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:

`Graph`, `DiGraph`, `MultiGraph`

Examples

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

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

G can be grown in several ways.

Nodes:

Add one node at a time:

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

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

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

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

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

Edges:

G can also be grown by adding edges.

Add one edge,

```
>>> 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
>>> print [n for n in G if n<3]   # iterate through nodes
[1, 2]
>>> print len(G)   # number of nodes in graph
5
>>> print 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.iteritems():
...         for key,eattr in keydict.iteritems():
...             if 'weight' in eattr:
...                 print (n,nbr,eattr['weight'])
(1, 2, 4)
```

```
(2, 3, 8)
>>> print [ (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.

Methods

<table>
<tr><td></td><td>MultiDiGraph.__init__(**attr[, data, name])</td><td>Initialize a graph with edges, nar</td></tr>
<tr><td></td><td>MultiDiGraph.add_node(n, **attr[, attr_dict])</td><td>Add a single node n and update r</td></tr>
<tr><td></td><td>MultiDiGraph.add_nodes_from(nodes, **attr)</td><td>Add multiple nodes.</td></tr>
<tr><td></td><td>MultiDiGraph.remove_node(n)</td><td>Remove node n.</td></tr>
<tr><td></td><td>MultiDiGraph.remove_nodes_from(nbunch)</td><td>Remove multiple nodes.</td></tr>
<tr><td></td><td>MultiDiGraph.add_edge(u, v, **attr[, key, ...])</td><td>Add an edge between u and v.</td></tr>
<tr><td>Adding and Removing Nodes and Edges</td><td>MultiDiGraph.add_edges_from(ebunch, **attr)</td><td>Add all the edges in ebunch.</td></tr>
<tr><td></td><td>MultiDiGraph.add_weighted_edges_from(ebunch, ...)</td><td>Add all the edges in ebunch as w specified weights.</td></tr>
<tr><td></td><td>MultiDiGraph.remove_edge(u, v[, key])</td><td>Remove the edge between u and</td></tr>
<tr><td></td><td>MultiDiGraph.remove_edges_from(ebunch)</td><td>Remove all edges specified in eb</td></tr>
<tr><td></td><td>MultiDiGraph.add_star(nlist, **attr)</td><td>Add a star.</td></tr>
<tr><td></td><td>MultiDiGraph.add_path(nlist, **attr)</td><td>Add a path.</td></tr>
<tr><td></td><td>MultiDiGraph.add_cycle(nlist, **attr)</td><td>Add a cycle.</td></tr>
<tr><td></td><td>MultiDiGraph.clear()</td><td>Remove all nodes and edges fror</td></tr>
</table>

networkx.MultiDiGraph.__init__

__init__(*data=None, name=", **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

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

add_nodes_from(*nodes, **attr*)

Add multiple nodes.

> **Parameters nodes** : iterable container
>
> > A container of nodes (list, dict, set, etc.). The container will be iterated through once.
>
> > **attr** : keyword arguments, optional (default= no attributes)
> >
> > > Update attributes for all nodes in nodes.

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())
[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)
```

networkx.MultiDiGraph.remove_node

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

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

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.MultiDiGraph.add_edges_from

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

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

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

Remove the edge between u and v.

Parameters **u,v: nodes** :

Remove edge or edges between nodes u and v.

> **key** : hashable identifier, optional (default= None)
>
>> Used to distinguish multiedges between a pair of nodes. If None, remove all edges
>> between 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.MultiGraph()   # or MultiDiGraph, 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
>>> G.remove_edge(2,3,key=0) # identify an individual edge with key
```

networkx.MultiDiGraph.remove_edges_from

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

networkx.MultiDiGraph.add_star

add_star(*nlist, **attr*)

> Add a star.
>
> The first node in nlist is the middle of the star. It is connected to all other nodes in nlist.
>
> > **Parameters nlist** : list
> >
> > > A list 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

add_path(*nlist, **attr*)

> Add a path.
>
> > **Parameters nlist** : list
> >
> > > A list 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

add_cycle(*nlist, **attr*)

> Add a cycle.
>
> > **Parameters nlist** : list
> >
> > > A list 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

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()
[]
```

	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, data])	Return a list of edges.
	MultiDiGraph.out_edges_iter([nbunch, data, keys])	Return an iterator over the edges.
	MultiDiGraph.in_edges([nbunch, data])	Return a list of the incoming edges.
	MultiDiGraph.in_edges_iter([nbunch, data, keys])	Return an iterator over the incoming edges.
Iterating over nodes and edges	MultiDiGraph.get_edge_data(u, v[, key, default])	Return the attribute dictionary associated with
	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
	MultiDiGraph.adjacency_iter()	Return an iterator of (node, adjacency dict) tu nodes.
	MultiDiGraph.nbunch_iter([nbunch])	Return an iterator of nodes contained in nbun in the graph.

networkx.MultiDiGraph.nodes

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

networkx.MultiDiGraph.nodes_iter

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'.

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

Examples

```python
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> for n in G.nodes_iter():
...     print n,
0 1 2
>>> for n,d in G.nodes_iter(data=True):
...     print d,
{} {} {}
```

networkx.MultiDiGraph.__iter__

__iter__ ()

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

Returns niter : iterator

An iterator over all nodes in the graph.

Examples

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

networkx.MultiDiGraph.edges

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).
>
> **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.MultiDiGraph.edges_iter

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.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.MultiDiGraph.out_edges

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.MultiDiGraph.out_edges_iter

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.

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.MultiDiGraph.in_edges

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

Return a list of the incoming edges.

See Also:

edges return a list of edges

networkx.MultiDiGraph.in_edges_iter

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

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.
> >
> > **Returns edge_dict** : dictionary
> >
> > > The edge attribute dictionary.

Notes

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

```
>>> G = nx.MultiGraph()
>>> G.add_path([0,1,2,3])
>>> G[0][1][0]
{}
```

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

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

Examples

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

neighbors(*n*)

Return a list of successor nodes of n.

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

networkx.MultiDiGraph.neighbors_iter

neighbors_iter(*n*)

Return an iterator over successor nodes of n.

neighbors_iter() and successors_iter() are the same.

networkx.MultiDiGraph.__getitem__

__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])
>>> print G[0]
{1: {}}
```

networkx.MultiDiGraph.successors

`successors`(*n*)

    Return a list of successor nodes of n.

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

networkx.MultiDiGraph.successors_iter

`successors_iter`(*n*)

    Return an iterator over successor nodes of n.

    neighbors_iter() and successors_iter() are the same.

networkx.MultiDiGraph.predecessors

`predecessors`(*n*)

    Return a list of predecessor nodes of n.

networkx.MultiDiGraph.predecessors_iter

`predecessors_iter`(*n*)

    Return an iterator over predecessor nodes of n.

networkx.MultiDiGraph.adjacency_list

`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

`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

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.

	MultiDiGraph.has_node(n)	Return True if the graph contains the n
	MultiDiGraph.__contains__(n)	Return True if n is a node, False other expression
	MultiDiGraph.has_edge(u, v[, key])	Return True if the edge (u,v) is in the g
	MultiDiGraph.nodes_with_selfloops()	Return a list of nodes with self loops.
	MultiDiGraph.selfloop_edges([data])	Return a list of selfloop edges.
	MultiDiGraph.order()	Return the number of nodes in the grap
Information about graph structure	MultiDiGraph.number_of_nodes()	Return the number of nodes in the grap
	MultiDiGraph.__len__()	Return the number of nodes.
	MultiDiGraph.size([weighted])	Return the number of edges.
	MultiDiGraph.number_of_edges([u, v])	Return the number of edges between tw
	MultiDiGraph.number_of_selfloops()	Return the number of selfloop edges.
	MultiDiGraph.degree([nbunch, with_labels, ...])	Return the degree of a node or nodes.
	MultiDiGraph.degree_iter([nbunch, weighted])	Return an iterator for (node, degree).

networkx.MultiDiGraph.has_node

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])
>>> print G.has_node(0)
True
```

It is more readable and simpler to use

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

networkx.MultiDiGraph.__contains__

__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])
>>> print 1 in G
True
```

networkx.MultiDiGraph.has_edge

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

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.MultiDiGraph.nodes_with_selfloops

`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

`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.MultiDiGraph.order

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

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])
>>> print len(G)
3
```

networkx.MultiDiGraph.__len__

__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])
>>> print len(G)
4
```

networkx.MultiDiGraph.size

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
```

networkx.MultiDiGraph.number_of_edges

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.number_of_selfloops

`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
```

networkx.MultiDiGraph.degree

`degree`(*nbunch=None, with_labels=False, 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.
>
> **with_labels** : bool, optional (default=False)
>
>> If True return a dictionary of degrees keyed by node.
>
> **weighted** : bool, optional (default=False)
>
>> If True return the sum of edge weights adjacent to the node.
>
> **Returns nd** : list, or dictionary
>
>> A list of node degrees or a dictionary with nodes as keys and degree as values if with_labels=True).

Examples

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

networkx.MultiDiGraph.degree_iter

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)]
```

Making copies and subgraphs	MultiDiGraph.copy()	Return a copy of the graph.
	MultiDiGraph.to_undirected()	Return an undirected representation of the digraph
	MultiDiGraph.subgraph(nbunch[, copy])	Return the subgraph induced on nodes in nbunch
	MultiDiGraph.reverse([copy])	Return the reverse of the graph.

networkx.MultiDiGraph.copy

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

`to_undirected`()

Return an undirected representation of the 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 is similar to MultiGraph(self) which returns a shallow copy. self.to_undirected() returns a deepcopy of edge, node and graph attributes.

networkx.MultiDiGraph.subgraph

`subgraph`(*nbunch, copy=True*)

Return the subgraph induced on nodes in nbunch.

The induced subgraph of the graph has the nodes in nbunch as its node set and the edges adjacent to those nodes as its edge set.

    Parameters nbunch : list, iterable

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

    copy : bool, optional (default=True)

        If True return a new graph holding the subgraph including copies of all edge and node properties. If False the subgraph is created using the original graph by deleting all nodes not in nbunch (this changes the original graph).

    Returns G : Graph

        A subgraph of the graph. If copy=True a new graph is returned with copies of graph, node, and edge data. If copy=False the subgraph is created in place by modifying the original graph.

Notes

If copy=True, nodes and edges are copied using copy.deepcopy.

Examples

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

networkx.MultiDiGraph.reverse

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).

3.2.2 Which graph class should I use?

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

3.3 Operators

Operations on graphs; including union, intersection, difference, complement, subgraph.

The following operator functions provide standard relabeling and combining operations for networks.

3.3.1 Graph Manipulation

create_empty_copy(G[, with_nodes])	Return a copy of the graph G with all of the edges removed.
subgraph(G, nbunch[, copy])	Return the subgraph induced on nodes in nbunch.
cartesian_product(G, H)	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
intersection(G, H[, create_using])	Return a new graph that contains only the edges that exist in
difference(G, H[, create_using])	Return a new graph that contains the edges that exist in
symmetric_difference(G, H[, create_using])	Return new graph with edges that exist in
line_graph(G)	Return the line graph of the graph or digraph G.
ego_graph(G, n[, center])	Returns induced subgraph of neighbors centered at node n.
stochastic_graph(G[, copy])	Return a right-stochastic representation of G.

networkx.create_empty_copy

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.

networkx.subgraph

subgraph (*G, nbunch, copy=True*)

Return the subgraph induced on nodes in nbunch.

> **Parameters** **G** : graph
>
>> A NetworkX graph
>
> **nbunch** : list, iterable
>
>> A container of nodes that will be iterated through once (thus it should be an iterator or be iterable). Each element of the container should be a valid node type: any hashable type except None. If nbunch is None, return all edges data in the graph. Nodes in nbunch that are not in the graph will be (quietly) ignored.
>
> **copy** : bool (default True)
>
>> If True return a new graph holding the subgraph. Otherwise, the subgraph is created in the original graph by deleting nodes not in nbunch. Warning: this can destroy the graph.

Notes

subgraph(G) calls G.subgraph()

networkx.cartesian_product

cartesian_product (*G, H*)

Return the Cartesian product of G and H.

> **Parameters G,H** : graph
>
>> A NetworkX graph

Notes

Only tested with Graph class.

networkx.compose

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.
>
> **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.

networkx.complement

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.

networkx.union

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.
>
> **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 name for union graph

Notes

To force a disjoint union with node relabeling, use disjoint_union(G,H) or convert_node_labels_to integers().

networkx.disjoint_union

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.

networkx.intersection

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
>
> **create_using** : NetworkX graph
>
> > Use specified graph for result. Otherwise a new graph is created.

Notes

If you want an new graph of the intersection of node sets of G and H with the edges from G use

```
>>> 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)
```

networkx.difference

difference(*G, H, create_using=None*)

Return a new graph that contains the edges that exist in in G but not in H.

The node sets of H and G must be the same.

> **Parameters** **G,H** : graph
>
> > A NetworkX graph
>
> **create_using** : NetworkX graph
>
> > Use specified graph for result. Otherwise a new graph is created.

Notes

If you want an new graph consisting of the difference of the node sets of G and H with the edges from G use

```
>>> 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)
```

networkx.symmetric_difference

symmetric_difference(*G, H, create_using=None*)

Return new graph with edges that exist in 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
>
> **create_using** : NetworkX graph
>
> > Use specified graph for result. Otherwise a new graph is created.

networkx.line_graph

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

See Also:

convert_node_labels_to_integers

Notes

Not implemented for MultiGraph or MultiDiGraph classes.

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))]
```

networkx.ego_graph

ego_graph (*G, n, center=True*)

> Returns induced subgraph of neighbors centered at node n.
>
> > **Parameters G** : graph
> >
> > > A NetworkX Graph or DiGraph
> >
> > **n** : node
> >
> > > A single node
> >
> > **center** : bool, optional
> >
> > > If False, do not include center node in graph

networkx.stochastic_graph

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

3.3.2 Node Relabeling

convert_node_labels_to_integers(G[, ...])	Return a copy of G node labels replaced with integers.
relabel_nodes(G, mapping)	Return a copy of G with node labels transformed by mapping.

networkx.convert_node_labels_to_integers

convert_node_labels_to_integers(*G, first_label=0, ordering='default', discard_old_labels=True*)

Return a copy of G node labels replaced with integers.

Parameters **G** : graph

A NetworkX graph

first_label : int, optional (default=0)

An integer specifying the offset in numbering nodes. The n new integer labels are numbered first_label, ..., n+first_label.

ordering : string

"default" : inherit node ordering from G.nodes() "sorted" : inherit node ordering from sorted(G.nodes()) "increasing degree" : nodes are sorted by increasing degree "decreasing degree" : nodes are sorted by decreasing degree

discard_old_labels : bool, optional (default=True)

if True (default) discard old labels if False, create a dict self.node_labels that maps new labels to old labels

networkx.relabel_nodes

relabel_nodes(*G, mapping*)

Return a copy of G with node labels transformed by mapping.

Parameters **G** : graph

A NetworkX graph

mapping : dictionary or function

Either a dictionary with the old labels as keys and new labels as values or a function transforming an old label with a new label. In either case, the new labels must be hashable Python objects.

See Also:

convert_node_labels_to_integers

Examples

mapping as dictionary

```
>>> G=nx.path_graph(3)  # nodes 0-1-2
>>> mapping={0:'a',1:'b',2:'c'}
>>> H=nx.relabel_nodes(G,mapping)
>>> print H.nodes()
['a', 'c', 'b']
```

```
>>> G=nx.path_graph(26) # nodes 0..25
>>> mapping=dict(zip(G.nodes(),"abcdefghijklmnopqrstuvwxyz"))
>>> H=nx.relabel_nodes(G,mapping) # nodes a..z
>>> mapping=dict(zip(G.nodes(),xrange(1,27)))
>>> G1=nx.relabel_nodes(G,mapping) # nodes 1..26
```

mapping as function

```
>>> G=nx.path_graph(3)
>>> def mapping(x):
...     return x**2
>>> H=nx.relabel_nodes(G,mapping)
>>> print H.nodes()
[0, 1, 4]
```

3.3.3 Freezing

freeze(G)	Modify graph to prevent addition of nodes or edges.
is_frozen(G)	Return True if graph is frozen"

networkx.freeze

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.path_graph(4)
>>> G=nx.freeze(G)
>>> G.add_edge(4,5)
...
NetworkXError: Frozen graph can't be modified
```

networkx.is_frozen

is_frozen(*G*)

Return True if graph is frozen"

> **Parameters G** : graph
>
> > A NetworkX graph
>
> **See Also:**
>
> > freeze

3.4 Algorithms

3.4.1 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. |

networkx.edge_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.

networkx.node_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.

3.4.2 Centrality

Centrality measures.

betweenness_centrality(G[, normalized, ...])	Compute betweenness centrality for nodes.
betweenness_centrality_source(G[, ...])	Compute betweenness centrality for a subgraph.
load_centrality(G[, v, cutoff, normalized, ...])	Compute load centrality for nodes.
edge_betweenness(G[, normalized, ...])	Compute betweenness centrality for edges.
degree_centrality(G[, v])	Compute the degree centrality for nodes.
closeness_centrality(G[, v, weighted_edges])	Compute closeness centrality for nodes.
eigenvector_centrality(G[, max_iter, tol, ...])	Compute the eigenvector centrality for the graph G.

networkx.betweenness_centrality

betweenness_centrality(*G, normalized=True, weighted_edges=False*)
Compute betweenness centrality for nodes.

Betweenness 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.
>
>> **weighted_edges** : bool, optional

> > Consider the edge weights in determining the shortest paths. If False, all edge weights
> > are considered equal.

> > **Returns nodes** : dictionary

> > > Dictionary of nodes with betweeness centrality as the value.

> **See Also:**

> `load_centrality`

Notes

The algorithm is from Ulrik Brandes, A Faster Algorithm for Betweenness Centrality. Journal of Mathematical Sociology 25(2):163-177, 2001. http://www.inf.uni-konstanz.de/algo/publications/b-fabc-01.pdf

networkx.betweenness_centrality_source

betweenness_centrality_source (*G, normalized=True, weighted_edges=False, sources=None*)
Compute betweenness centrality for a subgraph.

Enchanced version of the method in centrality module that allows specifying a list of sources (subgraph).

> **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.

> **weighted_edges** : bool, optional

> > Consider the edge weights in determining the shortest paths. If False, all edge weights are considered equal.

> **sources** : node list

> > A list of nodes to consider as sources for shortest paths.

> **Returns nodes** : dictionary

> > Dictionary of nodes with betweeness centrality as the value.

Notes

See Sec. 4 in Ulrik Brandes, A Faster Algorithm for Betweenness Centrality. Journal of Mathematical Sociology 25(2):163-177, 2001. http://www.inf.uni-konstanz.de/algo/publications/b-fabc-01.pdf

This algorithm does not count the endpoints, i.e. a path from s to t does not contribute to the betweenness of s and t.

networkx.load_centrality

load_centrality (*G, v=None, cutoff=None, normalized=True, weighted_edges=False*)
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.

weighted_edges : bool, optional

>   Consider the edge weights in determining the shortest paths. If False, all edge weights
>   are considered equal.

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.edge_betweenness

edge_betweenness(*G, normalized=True, weighted_edges=False, sources=None*)

>   Compute betweenness centrality for edges.

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.

weighted_edges : bool, optional

>   Consider the edge weights in determining the shortest paths. If False, all edge weights
>   are considered equal.

sources : node list

>   A list of nodes to consider as sources for shortest paths.

Returns nodes : dictionary

>   Dictionary of edges with betweeness centrality as the value.

networkx.degree_centrality

degree_centrality(*G, v=None*)

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

v : node, optional

Return only the value for node v.

Returns **nodes** : dictionary

Dictionary of nodes with degree centrality as the value.

See Also:

betweenness_centrality, load_centrality, eigenvector_centrality

Notes

The degree centrality is normalized to the maximum possible degree in the graph G. That is, G.degree(v)/(G.order()-1).

networkx.closeness_centrality

closeness_centrality(*G, v=None, weighted_edges=False*)

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.

weighted_edges : bool, optional

Consider the edge weights in determining the shortest paths. If False, all edge weights are considered equal.

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.

networkx.eigenvector_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 PageRank iteration for each node.
>
> **Returns** **nodes** : dictionary
>
>> Dictionary of nodes with eigenvector centrality as the value.

See Also:

pagerank

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().

3.4.3 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, with_labels, ...])	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.

networkx.find_cliques

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 recursive version of the same algorithm

Reference :

`Based`

http //doi.acm.org/10.1145/362342.362367

`as, Tanaka`

http //dx.doi.org/10.1016/j.tcs.2006.06.015

`and`

http //dx.doi.org/10.1016/j.tcs.2008.05.010

networkx.make_max_clique_graph

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.

networkx.make_clique_bipartite

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.

networkx.graph_clique_number

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.

networkx.graph_number_of_cliques

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.

networkx.node_clique_number

node_clique_number (*G, nodes=None, with_labels=False, cliques=None*)

Returns the size of the largest maximal clique containing each given node.

Returns a single or list depending on input nodes. Returns a dict keyed by node if "with_labels=True". Optional list of cliques can be input if already computed.

networkx.number_of_cliques

number_of_cliques (*G, nodes=None, cliques=None, with_labels=False*)

Returns the number of maximal cliques for each node.

Returns a single or list depending on input nodes. Returns a dict keyed by node if "with_labels=True". Optional list of cliques can be input if already computed.

networkx.cliques_containing_node

cliques_containing_node (*G, nodes=None, cliques=None, with_labels=False*)

Returns a list of cliques containing the given node.

Returns a single list or list of lists depending on input nodes. Returns a dict keyed by node if "with_labels=True". Optional list of cliques can be input if already computed.

3.4.4 Clustering

triangles(G[, nbunch, with_labels])	Compute the number of triangles.
transitivity(G)	Compute transitivity.
clustering(G[, nbunch, with_labels, weights])	Compute the clustering coefficient for nodes.
average_clustering(G)	Compute average clustering coefficient.

networkx.triangles

triangles(*G, nbunch=None, with_labels=False*)
  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
  >
  > **nbunch** : container of nodes, optional
  >
  > > Compute triangles for nodes in nbunch. The default is all nodes in G.
  >
  > **with_labels: bool, optional** :
  >
  > > If True return a dictionary keyed by node label.
  >
  > **Returns** **out** : list or dictionary
  >
  > > Number of trianges

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,with_labels=True)
{0: 6, 1: 6, 2: 6, 3: 6, 4: 6}
>>> print nx.triangles(G,(0,1))
[6, 6]
```

networkx.transitivity

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
```

networkx.clustering

clustering (*G, nbunch=None, with_labels=False, weights=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
>>
>> **nbunch** : container of nodes, optional
>>
>>> Limit to specified nodes. Default is entire graph.
>>
>> **with_labels: bool, optional** :
>>
>>> If True return a dictionary keyed by node label.
>>
>> **weights** : bool, optional
>>
>>> If True return fraction of connected triples as dictionary
>>
>> **Returns out** : float, list, dictionary or tuple of dictionaries
>>
>>> Clustering coefficient at specified nodes

Notes

The weights are the fraction of connected triples in the graph which include the keyed node. Ths is useful for computing transitivity.

Self loops are ignored.

Examples

```
>>> G=nx.complete_graph(5)
>>> print nx.clustering(G,0)
1.0
>>> print nx.clustering(G,with_labels=True)
{0: 1.0, 1: 1.0, 2: 1.0, 3: 1.0, 4: 1.0}
```

networkx.average_clustering

average_clustering(G)

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
>
> **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.

Examples

```
>>> G=nx.complete_graph(5)
>>> print nx.average_clustering(G)
1.0
```

3.4.5 Cores

Find and manipulate the k-cores of a graph

find_cores(G[, with_labels])	Return the core number for each vertex.

networkx.find_cores

find_cores(G, *with_labels=True*)

Return the core number for each vertex.

See: arXiv:cs.DS/0310049 by Batagelj and Zaversnik

If with_labels is True a dict is returned keyed by node to the core number. If with_labels is False a list of the core numbers is returned.

3.4.6 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 in the undirected, weighted graph G.

networkx.max_weight_matching

max_weight_matching(*G, maxcardinality=False*)
  Compute a maximum-weighted matching in the undirected, weighted graph G.

  If maxcardinality is True, compute the maximum-cardinality matching with maximum weight among all maximum-cardinality matchings.

  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.

  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.

  If G is an XGraph, the edge data are used as weight values; if G is a Graph, all edge weights are taken to be 1. Directed graphs and multi-edge graphs are not supported.

  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.

3.4.7 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.

networkx.is_isomorphic

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

networkx.could_be_isomorphic

`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.

networkx.fast_could_be_isomorphic

`fast_could_be_isomorphic`(*G1, G2*)

 Returns False if graphs are definitely not isomorphic. True does NOT guarantee isomorphism.

  Parameters G1, G2 : NetworkX graph instances

   The two graphs G1 and G2 must be the same type.

Notes

Checks for matching degree and triangle sequences.

networkx.faster_could_be_isomorphic

`faster_could_be_isomorphic`(*G1, G2*)

 Returns False if graphs are definitely not isomorphic. True does NOT guarantee isomorphism.

  Parameters G1, G2 : NetworkX graph instances

   The two graphs G1 and G2 must be the same type.

Notes

Checks for matching degree sequences.

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__

__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

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

is_isomorphic()

> Returns True if G1 and G2 are isomorphic graphs.

networkx.GraphMatcher.subgraph_is_isomorphic

subgraph_is_isomorphic()

> Returns True if a subgraph of G1 is isomorphic to G2.

networkx.GraphMatcher.isomorphisms_iter

isomorphisms_iter()

> Generator over isomorphisms between G1 and G2.

networkx.GraphMatcher.subgraph_isomorphisms_iter

subgraph_isomorphisms_iter()

> Generator over isomorphisms between a subgraph of G1 and G2.

networkx.GraphMatcher.candidate_pairs_iter

candidate_pairs_iter()

> Iterator over candidate pairs of nodes in G1 and G2.

networkx.GraphMatcher.match

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

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

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__

__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

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

is_isomorphic()

> Returns True if G1 and G2 are isomorphic graphs.

networkx.DiGraphMatcher.subgraph_is_isomorphic

subgraph_is_isomorphic()

> Returns True if a subgraph of G1 is isomorphic to G2.

networkx.DiGraphMatcher.isomorphisms_iter

isomorphisms_iter()

> Generator over isomorphisms between G1 and G2.

networkx.DiGraphMatcher.subgraph_isomorphisms_iter

`subgraph_isomorphisms_iter()`

> Generator over isomorphisms between a subgraph of G1 and G2.

networkx.DiGraphMatcher.candidate_pairs_iter

`candidate_pairs_iter()`

> Iterator over candidate pairs of nodes in G1 and G2.

networkx.DiGraphMatcher.match

`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

`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

`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__

__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

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

is_isomorphic ()

Returns True if G1 and G2 are isomorphic graphs.

networkx.WeightedGraphMatcher.subgraph_is_isomorphic

subgraph_is_isomorphic ()

Returns True if a subgraph of G1 is isomorphic to G2.

networkx.WeightedGraphMatcher.isomorphisms_iter

isomorphisms_iter ()

Generator over isomorphisms between G1 and G2.

networkx.WeightedGraphMatcher.subgraph_isomorphisms_iter

`subgraph_isomorphisms_iter()`
> Generator over isomorphisms between a subgraph of G1 and G2.

networkx.WeightedGraphMatcher.candidate_pairs_iter

`candidate_pairs_iter()`
> Iterator over candidate pairs of nodes in G1 and G2.

networkx.WeightedGraphMatcher.match

`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

`semantic_feasibility`(*G1_node, G2_node*)
> Returns True if mapping G1_node to G2_node is semantically feasible.

networkx.WeightedGraphMatcher.syntactic_feasibility

`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`()	Returns True if mapping G1_node to G2_node is semantically feasible.
`WeightedDiGraphMatcher.syntactic_feasibility`()	Returns True if adding (G1_node, G2_node) is syntactically feasible.

networkx.WeightedDiGraphMatcher.__init__

`__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

`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

`is_isomorphic()`
    Returns True if G1 and G2 are isomorphic graphs.

networkx.WeightedDiGraphMatcher.subgraph_is_isomorphic

`subgraph_is_isomorphic()`
    Returns True if a subgraph of G1 is isomorphic to G2.

networkx.WeightedDiGraphMatcher.isomorphisms_iter

`isomorphisms_iter()`
    Generator over isomorphisms between G1 and G2.

networkx.WeightedDiGraphMatcher.subgraph_isomorphisms_iter

`subgraph_isomorphisms_iter()`
    Generator over isomorphisms between a subgraph of G1 and G2.

networkx.WeightedDiGraphMatcher.candidate_pairs_iter

`candidate_pairs_iter()`
    Iterator over candidate pairs of nodes in G1 and G2.

networkx.WeightedDiGraphMatcher.match

`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

`semantic_feasibility(G1_node, G2_node)`
    Returns True if mapping G1_node to G2_node is semantically feasible.

networkx.WeightedDiGraphMatcher.syntactic_feasibility

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(...)	
WeightedMultiGraphMatcher.syntactic_feasibility(...)	Returns True if adding (G1_node, G2_node) is syntactically feasible.

networkx.WeightedMultiGraphMatcher.__init__

__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

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

is_isomorphic()

>   Returns True if G1 and G2 are isomorphic graphs.

networkx.WeightedMultiGraphMatcher.subgraph_is_isomorphic

`subgraph_is_isomorphic()`

>   Returns True if a subgraph of G1 is isomorphic to G2.

networkx.WeightedMultiGraphMatcher.isomorphisms_iter

`isomorphisms_iter()`

>   Generator over isomorphisms between G1 and G2.

networkx.WeightedMultiGraphMatcher.subgraph_isomorphisms_iter

`subgraph_isomorphisms_iter()`

>   Generator over isomorphisms between a subgraph of G1 and G2.

networkx.WeightedMultiGraphMatcher.candidate_pairs_iter

`candidate_pairs_iter()`

>   Iterator over candidate pairs of nodes in G1 and G2.

networkx.WeightedMultiGraphMatcher.match

`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

`semantic_feasibility`(*G1_node, G2_node*)

networkx.WeightedMultiGraphMatcher.syntactic_feasibility

`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__

__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

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

is_isomorphic()

Returns True if G1 and G2 are isomorphic graphs.

networkx.WeightedMultiDiGraphMatcher.subgraph_is_isomorphic

subgraph_is_isomorphic()

Returns True if a subgraph of G1 is isomorphic to G2.

networkx.WeightedMultiDiGraphMatcher.isomorphisms_iter

isomorphisms_iter()

Generator over isomorphisms between G1 and G2.

networkx.WeightedMultiDiGraphMatcher.subgraph_isomorphisms_iter

`subgraph_isomorphisms_iter()`
> Generator over isomorphisms between a subgraph of G1 and G2.

networkx.WeightedMultiDiGraphMatcher.candidate_pairs_iter

`candidate_pairs_iter()`
> Iterator over candidate pairs of nodes in G1 and G2.

networkx.WeightedMultiDiGraphMatcher.match

`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

`semantic_feasibility`(*G1_node, G2_node*)
> Returns True if mapping G1_node to G2_node is semantically feasible.

networkx.WeightedMultiDiGraphMatcher.syntactic_feasibility

`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.

3.4.8 PageRank

`pagerank`(G[, alpha, max_iter, tol, nstart])	Return the PageRank of the nodes in the graph.
`pagerank_numpy`(G[, alpha, max_iter, tol, ...])	Return a NumPy array of the PageRank of G.
`pagerank_scipy`(G[, alpha, max_iter, tol, ...])	Return a SciPy sparse array of the PageRank of G.
`google_matrix`(G[, alpha, nodelist])	

networkx.pagerank

`pagerank`(*G, alpha=0.84999999999999998, max_iter=100, tol=1e-08, nstart=None*)
> Return the PageRank of the nodes in the graph.

> PageRank computes the largest eigenvector of the stochastic adjacency matrix of G.

> **Parameters G** : graph
>> A networkx graph

>> **alpha** : float, optional
>>> Parameter for PageRank, default=0.85

>> **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 PageRank iteration for each node.

Returns nodes : dictionary

Dictionary of nodes with value as PageRank

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.

For an overview see: A. Langville and C. Meyer, "A survey of eigenvector methods of web information retrieval." http://citeseer.ist.psu.edu/713792.html

Examples

```
>>> G=nx.DiGraph(nx.path_graph(4))
>>> pr=nx.pagerank(G,alpha=0.9)
```

networkx.pagerank_numpy

pagerank_numpy(*G, alpha=0.84999999999999998, max_iter=100, tol=9.9999999999999995e-07, nodelist=None*)
Return a NumPy array of the PageRank of G.

networkx.pagerank_scipy

pagerank_scipy(*G, alpha=0.84999999999999998, max_iter=100, tol=9.9999999999999995e-07, nodelist=None*)
Return a SciPy sparse array of the PageRank of G.

networkx.google_matrix

google_matrix(*G, alpha=0.84999999999999998, nodelist=None*)

3.4.9 HITS

hits(G[, max_iter, tol, nstart])	Return hubs and authorities values.
hits_numpy(G[, max_iter, tol, nodelist])	Return a NumPy array of the hubs and authorities.
hits_scipy(G[, max_iter, tol, nodelist])	Return a SciPy sparse array of the hubs and authorities.
hub_matrix(G[, nodelist])	Return the HITS hub matrix.
authority_matrix(G[, nodelist])	Return the HITS authority matrix.

networkx.hits

hits(*G, max_iter=100, tol=1e-08, nstart=None*)
Return hubs and authorities values.

Uses the HITS algorithm.

    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-duple 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.

For an overview see: A. Langville and C. Meyer, "A survey of eigenvector methods of web information retrieval." http://citeseer.ist.psu.edu/713792.html

Examples

```
>>> G=nx.path_graph(4)
>>> h,a=nx.hits(G)
```

networkx.hits_numpy

hits_numpy(*G, max_iter=100, tol=9.9999999999999995e-07, nodelist=None*)
Return a NumPy array of the hubs and authorities.

networkx.hits_scipy

hits_scipy(*G, max_iter=100, tol=9.9999999999999995e-07, nodelist=None*)
Return a SciPy sparse array of the hubs and authorities.

networkx.hub_matrix

hub_matrix(*G, nodelist=None*)
>   Return the HITS hub matrix.

networkx.authority_matrix

authority_matrix(*G, nodelist=None*)
>   Return the HITS authority matrix.

3.4.10 Traversal

Components

Connected components and strongly connected components.

Undirected Graphs

is_connected(G)	Return True if G is connected.
number_connected_components(G)	Return the number of connected components in G.
connected_components(G)	Return a list of lists of nodes in each connected component of G.
connected_component_subgraphs(G)	Return a list of graphs of each connected component of G.
node_connected_component(G, n)	Return a list of nodes of the connected component containing node n.

networkx.is_connected
is_connected(*G*)
>   Return True if G is connected. For undirected graphs only.

networkx.number_connected_components
number_connected_components(*G*)
>   Return the number of connected components in G. For undirected graphs only.

networkx.connected_components
connected_components(*G*)
>   Return a list of lists of nodes in each connected component of G.
>
>   The list is ordered from largest connected component to smallest. For undirected graphs only.

networkx.connected_component_subgraphs
connected_component_subgraphs(*G*)
>   Return a list of graphs of each connected component of G.
>
>   The list is ordered from largest connected component to smallest. For undirected graphs only.

Examples

Get largest connected component

```python
>>> G=nx.path_graph(4)
>>> G.add_edge(5,6)
>>> H=nx.connected_component_subgraphs(G)[0]
```

networkx.node_connected_component

node_connected_component(*G, n*)

Return a list of nodes of the connected component containing node n.

For undirected graphs only.

Directed Graphs

is_strongly_connected(G)	Return True if G is strongly connected.
number_strongly_connected_components(G)	Return the number of strongly connected components in G.
strongly_connected_components(G)	Returns a list of strongly connected components in G.
strongly_connected_component_subgraphs(G)	Return a list of graphs of each strongly connected component of G.
strongly_connected_components_recursive(G)	Returns list of strongly connected components in G.
kosaraju_strongly_connected_components(G, ...])	Returns list of strongly connected components in G.

networkx.is_strongly_connected

is_strongly_connected(*G*)

Return True if G is strongly connected.

networkx.number_strongly_connected_components

number_strongly_connected_components(*G*)

Return the number of strongly connected components in G.

For directed graphs only.

networkx.strongly_connected_components

strongly_connected_components(*G*)

Returns a list of strongly connected components in G.

Uses Tarjan's algorithm with Nuutila's modifications. Nonrecursive version of algorithm.

References:

R. Tarjan (1972). Depth-first search and linear graph algorithms. SIAM Journal of Computing 1(2):146-160.

E. Nuutila and E. Soisalon-Soinen (1994). On finding the strongly connected components in a directed graph. Information Processing Letters 49(1): 9-14.

networkx.strongly_connected_component_subgraphs

strongly_connected_component_subgraphs (*G*)

> Return a list of graphs of each strongly connected component of G.
>
> The list is ordered from largest connected component to smallest.
>
> For example, to get the largest strongly connected component: >>> G=nx.path_graph(4) >>> H=nx.strongly_connected_component_subgraphs(G)[0]

networkx.strongly_connected_components_recursive

strongly_connected_components_recursive (*G*)

> Returns list of strongly connected components in G.
>
> Uses Tarjan's algorithm with Nuutila's modifications. this recursive version of the algorithm will hit the Python stack limit for large graphs.

networkx.kosaraju_strongly_connected_components

kosaraju_strongly_connected_components (*G, source=None*)

> Returns list of strongly connected components in G.
>
> Uses Kosaraju's algorithm.

DAGs

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)

networkx.topological_sort

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
> >
> > **nbunch** : container of nodes (optional)
> >
> > > Explore graph in specified order
>
> **See Also:**
>
> is_directed_acyclic_graph

Notes

If G is not a directed acyclic graph (DAG) no topological sort exists and the Python keyword None is returned.

This algorithm is based on a description and proof at http://www2.toki.or.id/book/AlgDesignManual/book/book2/node70.htm

networkx.topological_sort_recursive

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
>
>   **See Also:**
>
>   topological_sort, is_directed_acyclic_graph

Notes

>   If G is not a directed acyclic graph (DAG) no topological sort exists and the Python keyword None is returned.
>
>   This is a recursive version of topological sort.

networkx.is_directed_acyclic_graph

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

Distance

Shortest paths, diameter, radius, eccentricity, and related methods.

eccentricity(G[, v, sp, with_labels])	Return the eccentricity of node v in G (or all nodes if v is None).
diameter(G[, e])	Return the diameter of the graph G.
radius(G[, e])	Return the radius of the graph G.
periphery(G[, e])	Return the periphery of the graph G.
center(G[, e])	Return the center of graph G.

networkx.eccentricity

eccentricity(*G, v=None, sp=None, with_labels=False*)
>   Return the eccentricity of node v in G (or all nodes if v is None).
>
>   The eccentricity is the maximum of shortest paths to all other nodes.
>
>   The optional keyword sp must be a dict of dicts of shortest_path_length keyed by source and target. That is, sp[v][t] is the length from v to t.
>
>   If with_labels=True return dict of eccentricities keyed by vertex.

networkx.diameter

diameter (*G, e=None*)

> Return the diameter of the graph G.

> The diameter is the maximum of all pairs shortest path.

networkx.radius

radius (*G, e=None*)

> Return the radius of the graph G.

> The radius is the minimum of all pairs shortest path.

networkx.periphery

periphery (*G, e=None*)

> Return the periphery of the graph G.

> The periphery is the set of nodes with eccentricity equal to the diameter.

networkx.center

center (*G, e=None*)

> Return the center of graph G.

> The center is the set of nodes with eccentricity equal to radius.

Shortest Paths

Shortest path algorithms.

`average_shortest_path_length`(G[, weighted])	Return the average shortest path length.
`shortest_path`(G, source, target)	Return a list of nodes in a shortest path between source and target.
`shortest_path_length`(G, source, target)	Return the shortest path length between the source and target.
`single_source_shortest_path`(G, source[, cutoff])	Return list of nodes in a shortest path between source and all other nodes reachable from source.
`single_source_shortest_path_length`(G, source)	Return the shortest path length from source to all reachable nodes.
`all_pairs_shortest_path`(G[, cutoff])	Return shortest paths between all nodes.
`all_pairs_shortest_path_length`(G[, cutoff])	Return dictionary of shortest path lengths between all nodes in G.
`dijkstra_path`(G, source, target)	Returns the shortest path from source to target in a weighted
`dijkstra_path_length`(G, source, target)	Returns the shortest path length from source to target in a weighted
`single_source_dijkstra_path`(G, source)	Return list of nodes in a shortest path between source and all other nodes reachable from source for a weighted graph.
`single_source_dijkstra_path_length`(G, source)	Returns the shortest path lengths from source to all other
`single_source_dijkstra`(G, source[, target, ...])	Returns shortest paths and lengths in a weighted graph G.
`bidirectional_dijkstra`(G, source, target)	Dijkstra's algorithm for shortest paths using bidirectional search.
`bidirectional_shortest_path`(G, source, target)	Return a list of nodes in a shortest path between source and target.
`dijkstra_predecessor_and_distance`(G, source)	Returns two dictionaries representing a list of predecessors of a node and the distance to each node respectively.
`predecessor`(G, source[, target, cutoff, ...])	Returns dictionary of predecessors for the path from source to all
`floyd_warshall`(G[, huge])	The Floyd-Warshall algorithm for all pairs shortest paths.

networkx.average_shortest_path_length

average_shortest_path_length(*G, weighted=False*)

    Return the average shortest path length.

        Parameters G : NetworkX graph

            weighted : bool, optional, default=False

                If true use edge weights on path. If False, use 1 as the edge distance.

Examples

```
>>> G=nx.path_graph(4)
>>> print nx.average_shortest_path_length(G)
1.25
```

networkx.shortest_path

shortest_path(*G, source, target*)

Return a list of nodes in a shortest path between source and target.

There may be more than one shortest path. This returns only one.

> **Parameters** **G** : NetworkX graph
>
>> **source** : node label
>>
>>> starting node for path
>>
>> **target** : node label
>>
>>> ending node for path

Examples

```
>>> G=nx.path_graph(5)
>>> print nx.shortest_path(G,0,4)
[0, 1, 2, 3, 4]
```

networkx.shortest_path_length

shortest_path_length(*G, source, target*)

Return the shortest path length between the source and target.

Raise an exception if no path exists.

> **Parameters** **G** : NetworkX graph
>
>> **source** : node label
>>
>>> Starting node for path
>>
>> **target** : node label
>>
>>> Ending node for path

Notes

G is treated as an unweighted graph. For weighted graphs see dijkstra_path_length.

Examples

```
>>> G=nx.path_graph(5)
>>> print nx.shortest_path_length(G,0,4)
4
```

networkx.single_source_shortest_path

single_source_shortest_path(*G, source, cutoff=None*)
Return list of nodes in a shortest path between source and all other nodes reachable from source.

There may be more than one shortest path between the source and target nodes - this routine returns only one.

Returns a dictionary of shortest path lengths keyed by target.

> **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.

See Also:

shortest_path

Examples

```
>>> G=nx.path_graph(5)
>>> path=nx.single_source_shortest_path(G,0)
>>> path[4]
[0, 1, 2, 3, 4]
```

networkx.single_source_shortest_path_length

single_source_shortest_path_length(*G, source, cutoff=None*)
Return the shortest path length from source to all reachable nodes.

Returns a dictionary of shortest path lengths keyed by target.

> **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.

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.all_pairs_shortest_path

all_pairs_shortest_path (*G, cutoff=None*)

> Return shortest paths between all nodes.
>
> Returns a dictionary with keys for all reachable node pairs.
>
> > **Parameters** **G** : NetworkX graph
> >
> > > **cutoff** : integer, optional
> > >
> > > > depth to stop the search - only paths of length <= cutoff are returned.
>
> **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.all_pairs_shortest_path_length

all_pairs_shortest_path_length (*G, cutoff=None*)

> Return dictionary of shortest path lengths between all nodes in G.
>
> The dictionary only has keys for reachable node pairs.
>
> > **Parameters** **G** : NetworkX graph
> >
> > > **cutoff** : integer, optional
> > >
> > > > depth to stop the search - only paths of length <= cutoff are returned.

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.dijkstra_path

dijkstra_path (*G, source, target*)

> Returns the shortest path from source to target in a weighted graph G.
>
> Uses a bidirectional version of Dijkstra's algorithm.

> **Parameters** **G** : NetworkX graph
>
> > **source** : node label
> >
> > > starting node for path
> >
> > **target** : node label
> >
> > > ending node for path

See Also:

Notes

Edge data must be numerical values for Graph and DiGraphs.

Examples

```
>>> G=nx.path_graph(5)
>>> print nx.dijkstra_path(G,0,4)
[0, 1, 2, 3, 4]
```

networkx.dijkstra_path_length

dijkstra_path_length(*G, source, target*)

Returns the shortest path length from source to target in a weighted graph G.

Raise an exception if no path exists.

> **Parameters** **G** : NetworkX graph, weighted
>
> > **source** : node label
> >
> > > starting node for path
> >
> > **target** : node label
> >
> > > ending node for path

See Also:

Notes

Edge data must be numerical values for Graph and DiGraphs.

Examples

```
>>> G=nx.path_graph(5) # a weighted graph by default
>>> print nx.dijkstra_path_length(G,0,4)
4
```

networkx.single_source_dijkstra_path

single_source_dijkstra_path(*G, source*)

> Return list of nodes in a shortest path between source and all other nodes reachable from source for a weighted graph.
>
> Returns a dictionary of shortest path lengths keyed by target.
>
> > **Parameters** **G** : NetworkX graph
> >
> > > **source** : node label
> > >
> > > > starting node for path
>
> **See Also:**
>
> single_source_dijkstra

Notes

Edge data must be numerical values for Graph and DiGraphs.

Examples

```
>>> G=nx.path_graph(5)
>>> path=nx.single_source_dijkstra_path(G,0)
>>> path[4]
[0, 1, 2, 3, 4]
```

networkx.single_source_dijkstra_path_length

single_source_dijkstra_path_length(*G, source*)

> Returns the shortest path lengths from source to all other reachable nodes in a weighted graph G.
>
> Returns a dictionary of shortest path lengths keyed by target. Uses Dijkstra's algorithm.
>
> > **Parameters** **G** : NetworkX graph
> >
> > > **source** : node label
> > >
> > > > Starting node for path
>
> **See Also:**
>
> single_source_dijkstra

Notes

Edge data must be numerical values for XGraph and XDiGraphs.

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.single_source_dijkstra

single_source_dijkstra(*G, source, target=None, cutoff=None*)

Returns shortest paths and lengths in a weighted graph G.

Uses Dijkstra's algorithm for shortest paths. Returns a tuple of two dictionaries keyed by node. The first dicdtionary stores distance from the source. The second stores the path from the source to that node.

> **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.

See Also:

single_source_dijkstra_path, single_source_dijkstra_path_length

Notes

Distances are calculated as sums of weighted edges traversed. Edges must hold numerical values for Graph and DiGraphs.

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.bidirectional_dijkstra

bidirectional_dijkstra(*G, source, target*)

Dijkstra's algorithm for shortest paths using bidirectional search.

Returns a tuple of two dictionaries keyed by node. The first dicdtionary stores distance from the source. The second stores the path from the source to that node.

Raise an exception if no path exists.

> **Parameters** **G** : NetworkX graph
>
> > **source** : node label
> >
> > > Starting node for path
> >
> > **target** : node label
> >
> > > Ending node for path

Notes

Distances are calculated as sums of weighted edges traversed.

Edges must hold numerical values for Graph and DiGraphs.

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.bidirectional_shortest_path

bidirectional_shortest_path(*G, source, target*)

Return a list of nodes in a shortest path between source and target.

Also known as shortest_path()

> **Parameters** **G** : NetworkX graph
>
> > **source** : node label
> >
> > > starting node for path

target : node label

ending node for path

networkx.dijkstra_predecessor_and_distance

dijkstra_predecessor_and_distance(*G, source*)

Returns two dictionaries representing a list of predecessors of a node and the distance to each node respectively.

Parameters **G** : NetworkX graph

source : node label

Starting node for path

Notes

The list of predecessors contains more than one element only when there are more than one shortest paths to the key node.

This routine is intended for use with the betweenness centrality algorithms in centrality.py.

networkx.predecessor

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.

Examples

```
>>> G=nx.path_graph(4)
>>> print G.nodes()
[0, 1, 2, 3]
>>> nx.predecessor(G,0)
{0: [], 1: [0], 2: [1], 3: [2]}
```

networkx.floyd_warshall

floyd_warshall (*G, huge=inf*)
> The Floyd-Warshall algorithm for all pairs shortest paths.

> Returns a tuple (distance,path) containing two dictionaries of shortest distance and predecessor paths.

>> **Parameters   G** : NetworkX graph

> **See Also:**

> all_pairs_shortest_path, all_pairs_shortest_path_length

Notes

> This algorithm is most appropriate for dense graphs. The running time is O(n^3), and running space is O(n^2) where n is the number of nodes in G.

Shortest paths 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[, heuristic])	Return a list of nodes in a shortest path between source and target

networkx.astar_path

astar_path (*G, source, target, heuristic=None*)
> 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.

> **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((x1, y1), (x2, y2)):
```

```
...        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.astar_path_length

astar_path_length (*G, source, target, heuristic=None*)

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.

See Also:

astar_path

Search

Search algorithms.

See also networkx.path.

dfs_preorder(G[, source, reverse_graph])	Return list of nodes connected to source in depth-first-search preorder.
dfs_postorder(G[, source, reverse_graph])	Return list of nodes connected to source in depth-first-search postorder.
dfs_predecessor(G[, source, reverse_graph])	Return predecessors of depth-first-search with root at source.
dfs_successor(G[, source, reverse_graph])	Return succesors of depth-first-search with root at source.
dfs_tree(G[, source, reverse_graph])	Return directed graph (tree) of depth-first-search with root at source.

networkx.dfs_preorder

dfs_preorder (*G, source=None, reverse_graph=False*)

Return list of nodes connected to source in depth-first-search preorder.

Traverse the graph G with depth-first-search from source. Non-recursive algorithm.

networkx.dfs_postorder

dfs_postorder(*G, source=None, reverse_graph=False*)
>   Return list of nodes connected to source in depth-first-search postorder.
>
>   Traverse the graph G with depth-first-search from source. Non-recursive algorithm.

networkx.dfs_predecessor

dfs_predecessor(*G, source=None, reverse_graph=False*)
>   Return predecessors of depth-first-search with root at source.

networkx.dfs_successor

dfs_successor(*G, source=None, reverse_graph=False*)
>   Return succesors of depth-first-search with root at source.

networkx.dfs_tree

dfs_tree(*G, source=None, reverse_graph=False*)
>   Return directed graph (tree) of depth-first-search with root at source.
>
>   If the graph is disconnected, return a disconnected graph (forest).

3.4.11 Bipartite

is_bipartite(G)	Returns True if graph G is bipartite, False if not.
bipartite_sets(G)	Returns bipartite node sets of graph G.
bipartite_color(G)	Returns a two-coloring of the graph.
project(B, nodes[, create_using])	Return the projection of the graph onto a subset of nodes.

networkx.is_bipartite

is_bipartite(*G*)
>   Returns True if graph G is bipartite, False if not.
>
>   > **Parameters G** : NetworkX graph
>
>   **See Also:**
>
>   bipartite_color

Examples

```
>>> G=nx.path_graph(4)
>>> print nx.is_bipartite(G)
True
```

networkx.bipartite_sets

bipartite_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:

bipartite_color

Examples

```
>>> G=nx.path_graph(4)
>>> X,Y=nx.bipartite_sets(G)
>>> print X
set([0, 2])
>>> print Y
set([1, 3])
```

networkx.bipartite_color

bipartite_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.

Examples

```
>>> G=nx.path_graph(4)
>>> c=nx.bipartite_color(G)
>>> print c
{0: 1, 1: 0, 2: 1, 3: 0}
```

networkx.project

project(*B, nodes, create_using=None*)

Return the projection of the graph onto a subset of 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.

Returns Graph : NetworkX graph

A graph that is the projection onto the given nodes.

See Also:

is_bipartite, bipartite_sets

Notes

Returns a graph that is the projection of the bipartite graph B onto the set of nodes given in list nodes. No attempt is made to verify that the input graph B is bipartite.

Examples

```
>>> B=nx.path_graph(4)
>>> G=nx.project(B,[1,3])
>>> print G.nodes()
[1, 3]
>>> print G.edges()
[(1, 3)]
```

3.4.12 Minimum Spanning Tree

mst(G)	Generate a minimum spanning tree of an undirected graph.

networkx.mst

mst (*G*)

Generate a minimum spanning tree of an undirected graph.

Uses Kruskal's algorithm.

Parameters G : NetworkX Graph

Returns A generator that produces edges in the minimum spanning tree. :

The edges are three-tuples (u,v,w) where w is the weight. :

Notes

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.kruskal_mst(G) # a generator of MST edges
>>> edgelist=list(mst) # make a list of the edges
>>> print sorted(edgelist)
[(0, 1, {'weight': 1}), (1, 2, {'weight': 1}), (2, 3, {'weight': 1})]
>>> T=nx.Graph(edgelist)  # build a graph of the MST.
>>> print sorted(T.edges(data=True))
[(0, 1, {'weight': 1}), (1, 2, {'weight': 1}), (2, 3, {'weight': 1})]
```

3.5 Graph generators

3.5.1 Atlas

graph_atlas_g()	Return the list [G0,G1,...,G1252] of graphs as named in the Graph Atlas.

networkx.graph_atlas_g

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)

3.5.2 Classic

`balanced_tree`(r, h)	Return the perfectly balanced r-tree of height h.
`barbell_graph`(m1, m2)	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)	Return the complete bipartite graph K_{n1_n2}.
`circular_ladder_graph`(n)	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])	Return the 2d grid graph of mxn nodes, each connected to its nearest neighbors.
`grid_graph`(dim[, periodic])	Return the n-dimensional grid graph.
`hypercube_graph`(n)	Return the n-dimensional hypercube.
`ladder_graph`(n)	Return the Ladder graph of length n.
`lollipop_graph`(m, n)	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
`star_graph`(n)	Return the Star graph with n+1 nodes:
`trivial_graph`()	Return the Trivial graph with one node (with integer label 0)
`wheel_graph`(n)	Return the wheel graph: a single hub node connected to each node of the (n-1)-node cycle graph.

networkx.balanced_tree

balanced_tree(*r, h*)

Return the perfectly balanced r-tree of height h.

For r>=2, h>=1, 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.

number_of_nodes = 1+r+r**2+...+r**h = (r**(h+1)-1)/(r-1), number_of_edges = number_of_nodes - 1.

Node labels are the integers 0 (the root) up to number_of_nodes - 1.

networkx.barbell_graph

barbell_graph(*m1, m2*)

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.

networkx.complete_graph

complete_graph(*n, create_using=None*)

> Return the Complete graph K_n with n nodes.
>
> Node labels are the integers 0 to n-1.

networkx.complete_bipartite_graph

complete_bipartite_graph(*n1, n2*)

> 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

networkx.circular_ladder_graph

circular_ladder_graph(*n*)

> 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

networkx.cycle_graph

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.

networkx.dorogovtsev_goltsev_mendes_graph

dorogovtsev_goltsev_mendes_graph(*n*)

> Return the hierarchically constructed Dorogovtsev-Goltsev-Mendes graph.
>
> n is the generation. See: arXiv:/cond-mat/0112143 by Dorogovtsev, Goltsev and Mendes.

networkx.empty_graph

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).

networkx.grid_2d_graph

grid_2d_graph(*m, n, periodic=False*)
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.

networkx.grid_graph

grid_graph(*dim, periodic=False*)
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.

networkx.hypercube_graph

hypercube_graph(*n*)
Return the n-dimensional hypercube.

Node labels are the integers 0 to 2**n - 1.

networkx.ladder_graph

ladder_graph(*n*)
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.

networkx.lollipop_graph

lollipop_graph(*m, n*)
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.)

networkx.null_graph

null_graph (*create_using=None*)

Return the Null graph with no nodes or edges.

See empty_graph for the use of create_using.

networkx.path_graph

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.

networkx.star_graph

star_graph (*n*)

Return the Star graph with n+1 nodes: one center node, connected to n outer nodes.

Node labels are the integers 0 to n.

networkx.trivial_graph

trivial_graph ()

Return the Trivial graph with one node (with integer label 0) and no edges.

networkx.wheel_graph

wheel_graph (*n*)

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.

3.5.3 Small

make_small_graph(graph_description[, ...])	Return the small graph described by graph_description.
LCF_graph(n, shift_list, repeats)	Return the cubic graph specified in LCF notation.
bull_graph()	Return the Bull graph.
chvatal_graph()	Return the Chvatal graph.
cubical_graph()	Return the 3-regular Platonic Cubical graph.
desargues_graph()	Return the Desargues graph.
diamond_graph()	Return the Diamond graph.
dodecahedral_graph()	Return the Platonic Dodecahedral graph.
frucht_graph()	Return the Frucht Graph.
heawood_graph()	Return the Heawood graph, a (3,6) cage.
house_graph()	Return the House graph (square with triangle on top).
house_x_graph()	Return the House graph with a cross inside the house square.
icosahedral_graph()	Return the Platonic Icosahedral graph.
krackhardt_kite_graph()	Return the Krackhardt Kite Social Network.
moebius_kantor_graph()	Return the Moebius-Kantor graph.
octahedral_graph()	Return the Platonic Octahedral graph.
pappus_graph()	Return the Pappus graph.
petersen_graph()	Return the Petersen graph.
sedgewick_maze_graph()	Return a small maze with a cycle.
tetrahedral_graph()	Return the 3-regular Platonic Tetrahedral graph.
truncated_cube_graph()	Return the skeleton of the truncated cube.
truncated_tetrahedron_graph()	Return the skeleton of the truncated Platonic tetrahedron.
tutte_graph()	Return the Tutte graph.

networkx.make_small_graph

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 1,..,n.

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.

networkx.LCF_graph

LCF_graph(*n, shift_list, repeats*)

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.

networkx.bull_graph

bull_graph()

Return the Bull graph.

networkx.chvatal_graph

chvatal_graph()

Return the Chvatal graph.

networkx.cubical_graph

cubical_graph()

Return the 3-regular Platonic Cubical graph.

networkx.desargues_graph

desargues_graph()

Return the Desargues graph.

networkx.diamond_graph

diamond_graph()

Return the Diamond graph.

networkx.dodecahedral_graph

dodecahedral_graph()
> Return the Platonic Dodecahedral graph.

networkx.frucht_graph

frucht_graph()
> Return the Frucht Graph.

> The Frucht Graph is the smallest cubical graph whose automorphism group consists only of the identity element.

networkx.heawood_graph

heawood_graph()
> Return the Heawood graph, a (3,6) cage.

networkx.house_graph

house_graph()
> Return the House graph (square with triangle on top).

networkx.house_x_graph

house_x_graph()
> Return the House graph with a cross inside the house square.

networkx.icosahedral_graph

icosahedral_graph()
> Return the Platonic Icosahedral graph.

networkx.krackhardt_kite_graph

krackhardt_kite_graph()
> 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.

networkx.moebius_kantor_graph

moebius_kantor_graph()
> Return the Moebius-Kantor graph.

networkx.octahedral_graph

octahedral_graph()
> Return the Platonic Octahedral graph.

---

networkx.pappus_graph

pappus_graph()
> Return the Pappus graph.

networkx.petersen_graph

petersen_graph()
> Return the Petersen graph.

networkx.sedgewick_maze_graph

sedgewick_maze_graph()
> 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

networkx.tetrahedral_graph

tetrahedral_graph()
> Return the 3-regular Platonic Tetrahedral graph.

networkx.truncated_cube_graph

truncated_cube_graph()
> Return the skeleton of the truncated cube.

networkx.truncated_tetrahedron_graph

truncated_tetrahedron_graph()
> Return the skeleton of the truncated Platonic tetrahedron.

networkx.tutte_graph

tutte_graph()
> Return the Tutte graph.

3.5.4 Random Graphs

`fast_gnp_random_graph`(n, p[, seed])	Return a random graph G_{n,p}.
`gnp_random_graph`(n, p[, seed])	Return a random graph G_{n,p}.
`dense_gnm_random_graph`(n, m[, seed])	Return the random graph G_{n,m}.
`gnm_random_graph`(n, m[, seed])	Return the random graph G_{n,m}.
`erdos_renyi_graph`(n, p[, seed])	Return a random graph G_{n,p}.
`binomial_graph`(n, p[, seed])	Return a random graph G_{n,p}.
`newman_watts_strogatz_graph`(n, k, p[, seed])	Return a Newman-Watts-Strogatz small world graph.
`watts_strogatz_graph`(n, k, p[, seed])	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[, seed])	Return a random regular graph of n nodes each with degree d.
`barabasi_albert_graph`(n, m[, seed])	Return random graph using Barabási-Albert preferential
`powerlaw_cluster_graph`(n, m, p[, seed])	Holme and Kim algorithm for growing graphs with powerlaw
`random_lobster`(n, p1, p2[, seed])	Return a random lobster.
`random_shell_graph`(constructor[, seed])	Return a random shell graph for the constructor given.
`random_powerlaw_tree`(n[, gamma, seed, tries])	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.

networkx.fast_gnp_random_graph

fast_gnp_random_graph(*n, p, seed=None*)

Return a random graph G_{n,p}.

The G_{n,p} graph choses each of the possible [n(n-1)]/2 edges with probability p.

Sometimes called Erdős-Rényi graph, or binomial graph.

> **Parameters**
>
> - *n*: the number of nodes
>
> - *p*: probability for edge creation
>
> - *seed*: seed for random number generator (default=None)

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).

See:

Batagelj and Brandes, "Efficient generation of large random networks", Phys. Rev. E, 71, 036113, 2005.

networkx.gnp_random_graph

gnp_random_graph(*n, p, seed=None*)

Return a random graph G_{n,p}.

Choses each of the possible [n(n-1)]/2 edges with probability p. This is the same as binomial_graph and erdos_renyi_graph.

Sometimes called Erdős-Rényi graph, or binomial graph.

> **Parameters**
>
> > - *n*: the number of nodes
> >
> > - *p*: probability for edge creation
> >
> > - *seed*: seed for random number generator (default=None)

This is an O(n^2) algorithm. For sparse graphs (small p) see fast_gnp_random_graph.

P. Erdős and A. Rényi, On Random Graphs, Publ. Math. 6, 290 (1959). E. N. Gilbert, Random Graphs, Ann. Math. Stat., 30, 1141 (1959).

networkx.dense_gnm_random_graph

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*: the number of nodes
> >
> > - *m*: the number of edges
> >
> > - *seed*: seed for random number generator (default=None)

Algorithm by Keith M. Briggs Mar 31, 2006. Inspired by Knuth's Algorithm S (Selection sampling technique), in section 3.4.2 of

The Art of Computer Programming by Donald E. Knuth Volume 2 / Seminumerical algorithms Third Edition, Addison-Wesley, 1997.

networkx.gnm_random_graph

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.

> **Parameters**
>
> > - *n*: the number of nodes
> >
> > - *m*: the number of edges
> >
> > - *seed*: seed for random number generator (default=None)

networkx.erdos_renyi_graph

erdos_renyi_graph (*n, p, seed=None*)
  Return a random graph G_{n,p}.

Choses each of the possible [n(n-1)]/2 edges with probability p. This is the same as binomial_graph and erdos_renyi_graph.

Sometimes called Erdős-Rényi graph, or binomial graph.

> **Parameters**
>
> - *n*: the number of nodes
>
> - *p*: probability for edge creation
>
> - *seed*: seed for random number generator (default=None)

This is an O(n^2) algorithm. For sparse graphs (small p) see fast_gnp_random_graph.

P. Erdős and A. Rényi, On Random Graphs, Publ. Math. 6, 290 (1959). E. N. Gilbert, Random Graphs, Ann. Math. Stat., 30, 1141 (1959).

networkx.binomial_graph

binomial_graph(*n, p, seed=None*)
> Return a random graph G_{n,p}.

Choses each of the possible [n(n-1)]/2 edges with probability p. This is the same as binomial_graph and erdos_renyi_graph.

Sometimes called Erdős-Rényi graph, or binomial graph.

> **Parameters**
>
> - *n*: the number of nodes
>
> - *p*: probability for edge creation
>
> - *seed*: seed for random number generator (default=None)

This is an O(n^2) algorithm. For sparse graphs (small p) see fast_gnp_random_graph.

P. Erdős and A. Rényi, On Random Graphs, Publ. Math. 6, 290 (1959). E. N. Gilbert, Random Graphs, Ann. Math. Stat., 30, 1141 (1959).

networkx.newman_watts_strogatz_graph

newman_watts_strogatz_graph(*n, k, p, seed=None*)
> Return a Newman-Watts-Strogatz small world graph.

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.

> **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
>
> **seed** : int
>
> > seed for random number generator (default=None)

Notes

@ARTICLE{newman-1999-263, author = {M.~E.~J. Newman and D.~J. Watts}, title = {Renormalization group analysis of the small-world network model}, journal = {Physics Letters A}, volume = {263}, pages = {341}, url = {http://www.citebase.org/abstract?id=oai:arXiv.org:cond-mat/9903357}, year = {1999} }

networkx.watts_strogatz_graph

watts_strogatz_graph (*n, k, p, seed=None*)
    Return a Watts-Strogatz small-world graph.

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.

> **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
>
> **seed** : int
>
> > seed for random number generator (default=None)

See Also:

newman_watts_strogatz_graph, connected_watts_strogatz_graph

Notes

@article{Watts_Strogatz_1998, author = {Duncan J. Watts and Steven H. Strogatz}, title = {Collective dynamics of small-world networks}, journal = {Nature}, volume = {393}, pages = {440–442}, year = {1998}, }

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().

networkx.connected_watts_strogatz_graph

connected_watts_strogatz_graph (*n, k, p, tries=100, 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.

> **seed** : int

>> seed for random number generator (default=None)

See Also:

[newman_watts_strogatz_graph](#), [watts_strogatz_graph](#)

networkx.random_regular_graph

`random_regular_graph` (*d, n, 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** : integer

>>> Degree

>> **n** : integer

>>> Number of nodes. The nodes are numbered form 0 to n-1. The value of n*d must be even.

Notes

The algorithm is found in:

```
@misc{ steger-1999-generating,
author = "A. Steger and N. Wormald",
title = "Generating random regular graphs quickly",
text = "Probability and Computing 8 (1999), 377-396.",
year = "1999",
url = "citeseer.ist.psu.edu/steger99generating.html",
}
```

Kim and Vu's paper shows that this algorithm samples in an asymptotically uniform way from the space of random graphs when d = O(n**(1/3-epsilon)).

Reference:

```
@inproceedings{kim-2003-generating,
author = {Jeong Han Kim and Van H. Vu},
title = {Generating random regular graphs},
booktitle = {Proceedings of the thirty-fifth ACM symposium on
            Theory of computing},
year = {2003},
isbn = {1-58113-674-9},
pages = {213--222},
location = {San Diego, CA, USA},
```

```
doi = {http://doi.acm.org/10.1145/780542.780576},
publisher = {ACM Press},
}
```

networkx.barabasi_albert_graph

barabasi_albert_graph (*n, m, 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
>
> **seed** : hashable object (optional, default=None)
>
> > Seed for random number generator.
>
> **Returns**   **G** : Graph

Notes

The initialization is a graph with with m nodes and no edges.

References

[R1]

networkx.powerlaw_cluster_graph

powerlaw_cluster_graph (*n, m, p, seed=None*)
Holme and Kim algorithm for growing graphs with powerlaw degree distribution and approximate average clustering.

> **Parameters**
>
> > • *n*: the number of nodes
> >
> > • *m*: the number of random edges to add for each new node
> >
> > • *p*: probability of adding a triangle after adding a random edge
> >
> > • *seed*: seed for random number generator (default=None)

Reference:

```
@Article{growing-holme-2002,
author =         {P. Holme and B. J. Kim},
title =   {Growing scale-free networks with tunable clustering},
journal =        {Phys. Rev. E},
```

```
year =      {2002},
volume =            {65},
number =          {2},
pages =   {026107},
}
```

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 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 BA model.

networkx.random_lobster

random_lobster (*n, p1, p2, seed=None*)
    Return a random lobster.

> A caterpillar is a tree that reduces to a path graph when pruning all leaf nodes (p2=0). A lobster is a tree that reduces to a caterpillar when pruning all leaf nodes.

> **Parameters**
>
> - *n*: the expected number of nodes in the backbone
>
> - *p1*: probability of adding an edge to the backbone
>
> - *p2*: probability of adding an edge one level beyond backbone
>
> - *seed*: seed for random number generator (default=None)

networkx.random_shell_graph

random_shell_graph (*constructor, seed=None*)
    Return a random shell graph for the constructor given.

> •constructor: a list of three-tuples [(n1,m1,d1),(n2,m2,d2),..] one for each shell, starting at the center shell.
>
> •n : the number of nodes in the shell
>
> •m : the number or edges in the shell
>
> •**d** [the ratio of inter (next) shell edges to intra shell edges.] d=0 means no intra shell edges. d=1 for the last shell
>
> •*seed*: seed for random number generator (default=None)

```
>>> constructor=[(10,20,0.8),(20,40,0.8)]
>>> G=nx.random_shell_graph(constructor)
```

networkx.random_powerlaw_tree

random_powerlaw_tree (*n, gamma=3, seed=None, tries=100*)

Return a tree with a powerlaw degree distribution.

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).

Parameters

- *n*: the number of nodes

- *gamma*: exponent of power law is gamma

- *tries*: number of attempts to adjust sequence to make a tree

- *seed*: seed for random number generator (default=None)

networkx.random_powerlaw_tree_sequence

random_powerlaw_tree_sequence (*n, gamma=3, seed=None, tries=100*)

Return a degree sequence for a tree with a powerlaw distribution.

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).

Parameters

- *n*: the number of nodes

- *gamma*: exponent of power law is gamma

- *tries*: number of attempts to adjust sequence to make a tree

- *seed*: seed for random number generator (default=None)

3.5.5 Degree Sequence

configuration_model(deg_sequence[, seed])	Return a random graph with the given degree sequence.
expected_degree_graph(w[, seed])	Return a random graph G(w) with expected degrees given by w.
havel_hakimi_graph(deg_sequence)	Return a simple graph with given degree sequence, constructed using the
degree_sequence_tree(deg_sequence)	Make a tree for the given degree sequence.
is_valid_degree_sequence(deg_sequence)	Return True if deg_sequence is a valid sequence of integer degrees
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])	Attempt nswap double-edge swaps on the graph G.
connected_double_edge_swap(G, nswap])	Attempt nswap double-edge swaps on the graph G.
li_smax_graph(degree_seq)	Generates a graph based with a given degree sequence and maximizing the s-metric.
s_metric(G)	Return the "s-Metric" of graph G:

networkx.configuration_model

configuration_model(*deg_sequence, 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.

> > **seed** : hashable object (default=None)

> > > 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 [R2].

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

[R2]

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())
```

networkx.expected_degree_graph

expected_degree_graph (*w, seed=None*)
    Return a random graph G(w) with expected degrees given by w.

> **Parameters**
>
> > • *w*: a list of expected degrees
> >
> > • *seed*: seed for random number generator (default=None)

```
>>> z=[10 for i in range(100)]
>>> G=nx.expected_degree_graph(z)
```

Reference:

```
@Article{connected-components-2002,
  author =        {Fan Chung and L. Lu},
  title =         {Connected components in random graphs
  with given expected degree sequences},
  journal =       {Ann. Combinatorics},
  year =          {2002},
  volume =        {6},
  pages =         {125-145},
  }
```

networkx.havel_hakimi_graph

havel_hakimi_graph (*deg_sequence*)
    Return a simple graph with given degree sequence, constructed using the Havel-Hakimi algorithm.

> •*deg_sequence*: **degree sequence, a list of integers with each entry** corresponding to the degree of a
>   node (need not be sorted). A non-graphical degree sequence (not sorted). A non-graphical degree
>   sequence (i.e. one not realizable by some simple graph) raises an Exception.

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 [chartrand-graphs-1996]. This algorithm is also used in the function
is_valid_degree_sequence.

References:

[chartrand-graphs-1996] G. Chartrand and L. Lesniak, "Graphs and Digraphs", Chapman   and
    Hall/CRC, 1996.

networkx.degree_sequence_tree

degree_sequence_tree (*deg_sequence*)
    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

networkx.is_valid_degree_sequence

is_valid_degree_sequence(*deg_sequence*)

Return True if deg_sequence is a valid sequence of integer degrees equal to the degree sequence of some simple graph.

- •*deg_sequence*: **degree sequence, a list of integers with each entry** corresponding to the degree of a node (need not be sorted). A non-graphical degree sequence (i.e. one not realizable by some simple graph) will raise an exception.

See Theorem 1.4 in [chartrand-graphs-1996]. This algorithm is also used in havel_hakimi_graph()

References:

[chartrand-graphs-1996] G. Chartrand and L. Lesniak, "Graphs and Digraphs", Chapman and Hall/CRC, 1996.

networkx.create_degree_sequence

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).

- •*n*: length of degree sequence = number of nodes

- •*sfunction*: **a function, called as "sfunction(n,**kwds)",** that returns a list of n real or integer values.

- •*max_tries*: **max number of attempts at creating valid degree** sequence.

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.

```
>>> from networkx.utils import uniform_sequence
>>> seq=nx.create_degree_sequence(10,uniform_sequence)
```

networkx.double_edge_swap

double_edge_swap(*G, nswap=1*)

Attempt nswap double-edge swaps on the graph G.

Return count of successful swaps. The graph G is modified in place. 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

Does not enforce any connectivity constraints.

networkx.connected_double_edge_swap

connected_double_edge_swap(*G, nswap=1*)
> Attempt nswap double-edge swaps on the graph G.
>
> Returns count of successful swaps. Enforces connectivity. The graph G is modified in place.
>
> 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.
>
> Reference:
>
> ```
> @misc{gkantsidis-03-markov,
>  author = "C. Gkantsidis and M. Mihail and E. Zegura",
>  title = "The Markov chain simulation method for generating connected
>          power law random graphs",
>  year = "2003",
>  url = "http://citeseer.ist.psu.edu/gkantsidis03markov.html"
> }
> ```

networkx.li_smax_graph

li_smax_graph(*degree_seq*)
> 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.

networkx.s_metric

s_metric(G)

Return the "s-Metric" of graph G: the sum of the product deg(u)*deg(v) for every edge u-v in G

Reference:

```
@unpublished{li-2005,
 author = {Lun Li and David Alderson 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}
 }
```

3.5.6 Directed

gn_graph(n[, kernel, seed])	Return the GN (growing network) digraph with n nodes.
gnr_graph(n, p[, seed])	Return the GNR (growing network with redirection) digraph with n nodes
gnc_graph(n[, seed])	Return the GNC (growing network with copying) digraph with n nodes.
scale_free_graph(n[, G, alpha, beta, gamma, ...])	Return a scale free directed graph

networkx.gn_graph

gn_graph (*n, kernel=<function <lambda> at 0x897a6f4>, seed=None*)
Return the GN (growing network) digraph with n nodes.

The 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.

Example:

```
>>> 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
```

Reference:

```
@article{krapivsky-2001-organization,
title   = {Organization of Growing Random Networks},
author  = {P. L. Krapivsky and S. Redner},
journal = {Phys. Rev. E},
volume  = {63},
pages   = {066123},
year    = {2001},
}
```

networkx.gnr_graph

gnr_graph (*n, p, seed=None*)
Return the GNR (growing network with redirection) digraph with n nodes and redirection probability p.

The 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.

Example:

```
>>> D=nx.gnr_graph(10,0.5)   # the GNR graph
>>> G=D.to_undirected()   # the undirected version
```

Reference:

```
@article{krapivsky-2001-organization,
title   = {Organization of Growing Random Networks},
author  = {P. L. Krapivsky and S. Redner},
journal = {Phys. Rev. E},
volume  = {63},
pages   = {066123},
year    = {2001},
}
```

networkx.gnc_graph

gnc_graph (*n, seed=None*)

Return the GNC (growing network with copying) digraph with n nodes.

The 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.

Reference:

```
@article{krapivsky-2005-network,
title   = {Network Growth by Copying},
author  = {P. L. Krapivsky and S. Redner},
journal = {Phys. Rev. E},
volume  = {71},
pages   = {036118},
year    = {2005},
}
```

networkx.scale_free_graph

scale_free_graph (*n, G=None, alpha=0.40999999999999998, beta=0.54000000000000004, gamma=0.05000000000000003, delta_in=0.2000000000000001, delta_out=0, seed=None*)

Return a scale free directed graph

Parameters **n** : integer

Number of nodes in graph

G : NetworkX graph (optional)

Use as starting graph in algorithm

alpha : float

Probability for adding a new node conecgted 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.

delta_out : float

Bias for choosing ndoes from out-degree distribution.

seed : integer (optional)

Seed for random number generator

Notes

The sum of alpha, beta, and gamma must be 1.

Algorithm from

@article{bollobas2003dsf, title={{Directed scale-free graphs}}, author={Bollob{'a}s, B. and Borgs, C. and Chayes, J. and Riordan, O.}, journal={Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms}, pages={132–139}, year={2003}, publisher={Society for Industrial and Applied Mathematics Philadelphia, PA, USA} }

Examples

```
>>> G=nx.scale_free_graph(100)
```

3.5.7 Geometric

random_geometric_graph(n, radius[, ...])	Random geometric graph in the unit cube

networkx.random_geometric_graph

random_geometric_graph (*n, radius, create_using=None, repel=0.0, verbose=False, dim=2*)
    Random geometric graph in the unit cube

Returned Graph has added attribute G.pos which is a dict keyed by node to the position tuple for the node.

3.5.8 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

networkx.kl_connected_subgraph

kl_connected_subgraph(*G, k, l, low_memory=False, same_as_graph=False*)

Returns the maximum locally (k,l) connected subgraph of G.

(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

networkx.is_kl_connected

is_kl_connected(*G, k, l, low_memory=False*)

Returns True if G is kl connected

3.5.9 Bipartite

`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
`bipartite_reverse_havel_hakimi_g`(aseq, bseq)	Return a bipartite graph from two given degree sequences
`bipartite_alternating_havel_hakimi_g`(...)	Return a bipartite graph from two given degree sequences
`bipartite_preferential_attachment`(aseq, p)	Create a bipartite graph with a preferential attachment model from a given single degree sequence.
`bipartite_random_regular_graph`(d, n[, ...])	UNTESTED: Generate a random bipartite graph.

networkx.bipartite_configuration_model

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.

networkx.bipartite_havel_hakimi_graph

bipartite_havel_hakimi_graph(*aseq, bseq, create_using=None*)
    Return a bipartite graph from two given degree sequences using a Havel-Hakimi style construction.

        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.

        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. :

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.

networkx.bipartite_reverse_havel_hakimi_graph

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.

        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.

        Nodes from the 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. :

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.

networkx.bipartite_alternating_havel_hakimi_graph

bipartite_alternating_havel_hakimi_graph (*aseq, bseq, create_using=None*)
    Return a bipartite graph from two given degree sequences using a alternating Havel-Hakimi style construction.

        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.

         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. :

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.

networkx.bipartite_preferential_attachment_graph

bipartite_preferential_attachment_graph (*aseq, p, create_using=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.

Notes

@**article{guillaume-2004-bipartite,** author = {Jean-Loup Guillaume and Matthieu Latapy}, title = {Bipartite structure of all complex networks}, journal = {Inf. Process. Lett.}, volume = {90}, number = {5}, year = {2004}, issn = {0020-0190}, pages = {215–221}, doi = {http://dx.doi.org/10.1016/j.ipl.2004.03.007}, publisher = {Elsevier North-Holland, Inc.}, address = {Amsterdam, The Netherlands, The Netherlands}, }

networkx.bipartite_random_regular_graph

bipartite_random_regular_graph(*d, n, create_using=None*)
UNTESTED: Generate a random 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.

Notes

Nodes are numbered 0...n-1.

Restrictions on n and d:

- n must be even

- n>=2*d

Algorithm inspired by random_regular_graph()

3.6 Linear Algebra

3.6.1 Spectrum

`adj_matrix`(G[, nodelist])	Return adjacency matrix of graph as a numpy matrix.
`laplacian`(G[, nodelist])	Return standard combinatorial Laplacian of G as a numpy matrix.
`normalized_laplacian`(G[, nodelist])	Return normalized Laplacian of G as a numpy matrix.
`laplacian_spectrum`(G)	Return eigenvalues of the Laplacian of G
`adjacency_spectrum`(G)	Return eigenvalues of the adjacency matrix of G

networkx.adj_matrix

adj_matrix(*G, nodelist=None*)
Return adjacency matrix of graph as a numpy matrix.

This just calls networkx.convert.to_numpy_matrix.

If you want a pure python adjacency matrix represntation try networkx.convert.to_dict_of_dicts with weighted=False, which will return a dictionary-of-dictionaries format that can be addressed as a sparse matrix.

networkx.laplacian

laplacian(*G, nodelist=None*)
   Return standard combinatorial Laplacian of G as a numpy matrix.

   Return the matrix L = D - A, where

   D is the diagonal matrix in which the i'th entry is the degree of node i A is the adjacency matrix.

networkx.normalized_laplacian

normalized_laplacian(*G, nodelist=None*)
   Return normalized Laplacian of G as a numpy matrix.

   See Spectral Graph Theory by Fan Chung-Graham. CBMS Regional Conference Series in Mathematics, Number 92, 1997.

networkx.laplacian_spectrum

laplacian_spectrum(*G*)
   Return eigenvalues of the Laplacian of G

networkx.adjacency_spectrum

adjacency_spectrum(*G*)
   Return eigenvalues of the adjacency matrix of G

3.7 Converting to/from Other Formats

3.7.1 Convert

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 from_whatever() function which attempts to guess the input type and convert it automatically.

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.from_whatever(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)
```

See Also

For graphviz dot formats see networkx.drawing.nx_pygraphviz or networkx.drawing.nx_pydot.

3.7.2 Functions

from_whatever(thing[, create_using, ...])	Make a NetworkX graph from an known type.
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_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.
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.
to_numpy_matrix(G[, edge_attr, nodelist, dtype])	Return the graph adjacency matrix as a NumPy matrix.
from_numpy_matrix(A[, create_using])	Return a graph from numpy matrix adjacency list.
to_scipy_sparse_matrix(G[, edge_attr, ...])	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.

networkx.from_whatever

from_whatever(*thing, create_using=None, multigraph_input=False*)

Make a NetworkX graph from an known type.

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** **thing** : 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 thing is a dict_of_dicts, try to create a multigraph assuming dict_of_dict_of_lists. If thing and create_using are both multigraphs then create a multigraph from a multigraph.

networkx.to_dict_of_lists

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.

networkx.from_dict_of_lists

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

networkx.to_dict_of_dicts

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).

networkx.from_dict_of_dicts

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

networkx.to_edgelist

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

networkx.from_edgelist

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

networkx.to_numpy_matrix

to_numpy_matrix(*G, edge_attr='weight', nodelist=None, dtype=None*)

Return the graph adjacency matrix as a NumPy matrix.

> **Parameters G** : graph
>
>> A NetworkX graph
>
> **edge_attr** : None | str, optional
>
>> The attribute to use when populating the matrix. If None, then the values will appear as 1 or 0, depending on if the edge exists or not. For multigraphs, the value is represented by the sum of the edge attributes for each edge between the nodes. Thus, when edge_attr is None, the matrix elements represent the number of edges between the two nodes.
>
> **nodelist** : list, optional
>
>> Use the order of nodes in nodelist for the rows. If nodelist is None, the ordering is produced by G.nodes()
>
> **dtype** : numpy type
>
>> Data type for matrix entries. Default is 4 byte float (single).
>
> **Returns A** : NumPy matrix
>
>> Matrix constructed from edge_attr.

Examples

```
>>> G=nx.path_graph(4)
>>> A=nx.to_numpy_matrix(G)
```

networkx.from_numpy_matrix

from_numpy_matrix(*A, create_using=None*)

Return a graph from numpy matrix adjacency list.

> **Parameters A** : numpy matrix
>
>> An adjacency matrix representation of a graph
>
> **create_using** : NetworkX graph
>
>> Use specified graph for result. The default is Graph()

Examples

```
>>> import numpy
>>> A=numpy.matrix([[1,1],[2,1]])
>>> G=nx.from_numpy_matrix(A)
```

networkx.to_scipy_sparse_matrix

to_scipy_sparse_matrix(*G, edge_attr='weight', nodelist=None, dtype=None*)

    Return the graph adjacency matrix as a SciPy sparse matrix.

        Parameters G : graph

            A NetworkX graph

        edge_attr : None | str, optional

            The attribute to use when populating the matrix. If None, then the values will appear as 1 or 0, depending on if the edge exists or not. For multigraphs, the value is represented by the sum of the edge attributes for each edge between the nodes. Thus, when edge_attr is None, the matrix elements represent the number of edges between the two nodes.

        nodelist : list, optional

            Use the order of nodes in nodelist for the rows. If nodelist is None, the ordering is produced by G.nodes()

        dtype : numpy type

            Data type for matrix entries. Default is 4 byte float (single).

        Returns m : SciPy sparse matrix

            Matrix constructed from edge_attr.

Notes

Uses lil_matrix format. To convert to other formats see scipy.sparse documentation.

Examples

```
>>> G=nx.path_graph(4)
>>> A=nx.to_scipy_sparse_matrix(G)
>>> C=A.tocsr() # convert to compressed row storage
```

networkx.from_scipy_sparse_matrix

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)
```

3.8 Reading and Writing

3.8.1 Adjacency List

Read and write NetworkX graphs as adjacency lists.

Note that NetworkX graphs can contain any hashable Python object as node (not just integers and strings). So writing a NetworkX graph as a text file may not always be what you want: see write_gpickle and gread_gpickle for that case.

This module provides the following :

Adjacency list with single line per node: Useful for connected or unconnected graphs without edge data.

write_adjlist(G, path) G=read_adjlist(path)

Adjacency list with multiple lines per node: Useful for connected or unconnected graphs with or without edge data.

write_multiline_adjlist(G, path) read_multiline_adjlist(path)

read_adjlist(path[, comments, delimiter, ...])	Read graph in single line adjacency list format from path.
write_adjlist(G, path[, comments, delimiter])	Write graph G in single-line adjacency-list format to path.
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 the file

networkx.read_adjlist

read_adjlist(*path, comments='#', delimiter=' ', create_using=None, nodetype=None*)
    Read graph in single line adjacency list format from path.

Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_adjlist(G, "test.adjlist")
>>> G=nx.read_adjlist("test.adjlist")
```

path can be a filehandle or a string with the name of the file.

```
>>> fh=open("test.adjlist")
>>> 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")
```

nodetype is an optional 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.)

create_using is an optional networkx graph type, the default is Graph(), an undirected graph.

```
>>> G=nx.read_adjlist("test.adjlist", create_using=nx.DiGraph())
```

Does not handle edge data: use 'read_edgelist' or 'read_multiline_adjlist'

The comments character (default='#') at the beginning of a line indicates a comment line.

The entries are separated by delimiter (default=' '). If whitespace is significant in node or edge labels you should use some other delimiter such as a tab or other symbol.

Sample format:

```
# source target
a b c
d e
```

networkx.write_adjlist

write_adjlist(*G, path, comments='#', delimiter=' '*)
    Write graph G in single-line adjacency-list format to path.

    See read_adjlist for file format details.

Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_adjlist(G,"test.adjlist")
```

path can be a filehandle or a string with the name of the file.

```
>>> fh=open("test.adjlist",'w')
>>> nx.write_adjlist(G, fh)
```

Filenames ending in .gz or .bz2 will be compressed.

```
>>> nx.write_adjlist(G, "test.adjlist.gz")
```

The file will use the default text encoding on your system. It is possible to write files in other encodings by opening the file with the codecs module. See doc/examples/unicode.py for hints.

```
>>> import codecs
```

fh=codecs.open("test.adjlist",encoding='utf8') # use utf-8 encoding nx.write_adjlist(G,fh)

Does not handle edge data. Use 'write_edgelist' or 'write_multiline_adjlist'

networkx.read_multiline_adjlist

read_multiline_adjlist(*path, comments='#', delimiter=' ', create_using=None, nodetype=None, edgetype=None*)
Read graph in multi-line adjacency list format from path.

Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_multiline_adjlist(G,"test.adjlist")
>>> G=nx.read_multiline_adjlist("test.adjlist")
```

path can be a filehandle or a string with the name of the file.

```
>>> fh=open("test.adjlist")
>>> 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")
```

nodetype is an optional 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

Since nodes must be hashable, the function nodetype must return hashable types (e.g. int, float, str, frozenset - or tuples of those, etc.)

edgetype is a function to convert edge data strings to edgetype

```
>>> G=nx.read_multiline_adjlist("test.adjlist")
```

create_using is an optional networkx graph type, the default is Graph(), a simple undirected graph

```
>>> G=nx.read_multiline_adjlist("test.adjlist", create_using=nx.DiGraph())
```

The comments character (default='#') at the beginning of a line indicates a comment line.

The entries are separated by delimiter (default=' '). If whitespace is significant in node or edge labels you should use some other delimiter such as a tab or other symbol.

Example multiline adjlist file format

No edge data:

```
# source target for Graph or DiGraph
a 2
b
c
d 1
e
```

Wiht edge data::

```
# source target for XGraph or XDiGraph with edge data
a 2
b edge-ab-data
c edge-ac-data
d 1
e edge-de-data
```

Reading the file will use the default text encoding on your system. It is possible to read files with other encodings by opening the file with the codecs module. See doc/examples/unicode.py for hints.

```python
>>> import codecs
>>> fh=codecs.open("test.adjlist",'r',encoding='utf=8') # utf-8 encoding
>>> G=nx.read_multiline_adjlist(fh)
```

networkx.write_multiline_adjlist

write_multiline_adjlist (*G, path, delimiter=' ', comments='#'*)

Write the graph G in multiline adjacency list format to the file or file handle path.

See read_multiline_adjlist for file format details.

Examples

```python
>>> G=nx.path_graph(4)
>>> nx.write_multiline_adjlist(G,"test.adjlist")
```

path can be a filehandle or a string with the name of the file.

```python
>>> fh=open("test.adjlist",'w')
>>> nx.write_multiline_adjlist(G,fh)
```

Filenames ending in .gz or .bz2 will be compressed.

```python
>>> nx.write_multiline_adjlist(G,"test.adjlist.gz")
```

The file will use the default text encoding on your system. It is possible to write files in other encodings by opening the file with the codecs module. See doc/examples/unicode.py for hints.

```python
>>> import codecs
>>> fh=codecs.open("test.adjlist",'w',encoding='utf=8') # utf-8 encoding
>>> nx.write_multiline_adjlist(G,fh)
```

3.8.2 Edge List

Read and write NetworkX graphs as edge lists.

read_edgelist(path[, comments, delimiter, ...])	Read a graph from a list of edges.
write_edgelist(G, path[, comments, delimiter])	Write graph as a list of edges.

networkx.read_edgelist

read_edgelist(*path, comments='#', delimiter=' ', create_using=None, nodetype=None, edgetype=None*)

  Read a graph from a list of edges.

> **Parameters** **path** : file or string
>
> > File or filename to write. 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 uses to separate values. The default is whitespace.
>
> **create_using** : Graph container, optional
>
> > Use specified Graph container to build graph. The default is nx.Graph().
>
> **nodetype** : int, float, str, Python type, optional
>
> > Convert node data from strings to specified type
>
> **edgetype** : int, float, str, Python type, optional
>
> > Convert edge data from strings to specified type and use as 'weight'
>
> **Returns** **out** : 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 formats

Without edge data:

```
# source target
a b
a c
d e
```

With edge data::

```
# source target data
a b 1
a c 3.14159
d e apple
```

Examples

```
>>> nx.write_edgelist(nx.path_graph(4), "test.edgelist")
>>> G=nx.read_edgelist("test.edgelist")
```

```
>>> fh=open("test.edgelist")
>>> G=nx.read_edgelist(fh)
```

```
>>> G=nx.read_edgelist("test.edgelist", nodetype=int)
```

```
>>> G=nx.read_edgelist("test.edgelist",create_using=nx.DiGraph())
```

networkx.write_edgelist

write_edgelist(*G, path, comments='#', delimiter=' '*)

Write graph as a list of 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.
>
>   **comments** : string, optional
>
>>   The character used to indicate the start of a comment
>
>   **delimiter** : string, optional
>
>>   The string uses to separate values. The default is whitespace.

See Also:

networkx.write_edgelist

Notes

The file will use the default text encoding on your system. It is possible to write files in other encodings by opening the file with the codecs module. See doc/examples/unicode.py for hints.

```
>>> import codecs
>>> fh=codecs.open("test.edgelist",'w',encoding='utf=8') # utf-8 encoding
>>> nx.write_edgelist(G,fh)
```

Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_edgelist(G, "test.edgelist")
```

```
>>> fh=open("test.edgelist",'w')
>>> nx.write_edgelist(G,fh)
```

```
>>> nx.write_edgelist(G, "test.edgelist.gz")
```

3.8.3 GML

GML

Read graphs in GML 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)	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)	Parse GML format from string or iterable.

networkx.read_gml

read_gml(*path*)
> Read graph in GML format from path. Returns an Graph or DiGraph.
>
> This doesn't implement the complete GML specification for nested attributes for graphs, edges, and nodes.

networkx.write_gml

write_gml(*G, path*)
> Write the graph G in GML format to the file or file handle path.

Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_gml(G,"test.gml")
```

path can be a filehandle or a string with the name of the file.

```
>>> fh=open("test.gml",'w')
>>> nx.write_gml(G,fh)
```

Filenames ending in .gz or .bz2 will be compressed.

```
>>> nx.write_gml(G,"test.gml.gz")
```

The output file will use the default text encoding on your system. It is possible to write files in other encodings by opening the file with the codecs module. See doc/examples/unicode.py for hints.

```
>>> import codecs
>>> fh=codecs.open("test.gml",'w',encoding='iso8859-1')# use iso8859-1
>>> nx.write_gml(G,fh)
```

GML specifications indicate that the file should only use 7bit ASCII text encoding.iso8859-1 (latin-1).

Only a single level of attributes for graphs, nodes, and edges, is supported.

networkx.parse_gml

parse_gml(*lines*)
> Parse GML format from string or iterable. Returns an Graph or DiGraph.

> This doesn't implement the complete GML specification for nested attributes for graphs, edges, and nodes.

3.8.4 Pickle

Read and write NetworkX graphs as Python pickles.

Note that NetworkX graphs can contain any hashable Python object as node (not just integers and strings). So writing a NetworkX graph as a text file may not always be what you want: see write_gpickle and gread_gpickle for that case.

This module provides the following :

Python pickled format: Useful for graphs with non text representable data.

> write_gpickle(G, path) read_gpickle(path)

read_gpickle(path)	Read graph object in Python pickle format
write_gpickle(G, path)	Write graph object in Python pickle format.

networkx.read_gpickle

read_gpickle(*path*)
> Read graph object in Python pickle format

> G=nx.path_graph(4) nx.write_gpickle(G,"test.gpickle") G=nx.read_gpickle("test.gpickle")

> See cPickle.

networkx.write_gpickle

write_gpickle(*G, path*)
> Write graph object in Python pickle format.

> This will preserve Python objects used as nodes or edges.

> G=nx.path_graph(4) nx.write_gpickle(G,"test.gpickle")

> See cPickle.

3.8.5 GraphML

Read and write graphs in GraphML format. http://graphml.graphdrawing.org/

The module currently supports simple graphs and not nested graphs or hypergraphs.

read_graphml(path)	Read graph in GraphML format from path.
parse_graphml(lines)	Read graph in GraphML format from string.

networkx.read_graphml

read_graphml (*path*)
    Read graph in GraphML format from path.

    Returns a Graph or DiGraph.

    Does not implement full GraphML specification.

networkx.parse_graphml

parse_graphml (*lines*)
    Read graph in GraphML format from string.

    Returns a Graph or DiGraph.

    Does not implement full GraphML specification.

3.8.6 LEDA

Read graphs in LEDA format. See http://www.algorithmic-solutions.info/leda_guide/graphs/leda_native_graph_fileformat.html

read_leda(path)	Read graph in GraphML format from path.
parse_leda(lines)	Parse LEDA.GRAPH format from string or iterable.

networkx.read_leda

read_leda (*path*)
    Read graph in GraphML format from path. Returns an XGraph or XDiGraph.

networkx.parse_leda

parse_leda (*lines*)
    Parse LEDA.GRAPH format from string or iterable. Returns an Graph or DiGraph.

3.8.7 YAML

Read and write NetworkX graphs in YAML format. See http://www.yaml.org for documentation.

read_yaml(path)	Read graph from YAML format from path.
write_yaml(G, path, **kwds[, default_flow_style])	Write graph G in YAML text format to path.

networkx.read_yaml

read_yaml (*path*)
    Read graph from YAML format from path.

    See http://www.yaml.org

networkx.write_yaml

write_yaml(*G, path, default_flow_style=False, **kwds*)
> Write graph G in YAML text format to path.

> See http://www.yaml.org

3.8.8 SparseGraph6

Read graphs in graph6 and sparse6 format. See http://cs.anu.edu.au/~bdm/data/formats.txt

read_graph6(path)	Read simple undirected graphs in graph6 format from path.
parse_graph6(str)	Read undirected graph in graph6 format.
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(str)	Read undirected graph in sparse6 format.
read_sparse6_list(path)	Read simple undirected graphs in sparse6 format from path.

networkx.read_graph6

read_graph6(*path*)
> Read simple undirected graphs in graph6 format from path. Returns a single Graph.

networkx.parse_graph6

parse_graph6(*str*)
> Read undirected graph in graph6 format.

networkx.read_graph6_list

read_graph6_list(*path*)
> Read simple undirected graphs in graph6 format from path. Returns a list of Graphs, one for each line in file.

networkx.read_sparse6

read_sparse6(*path*)
> Read simple undirected graphs in sparse6 format from path. Returns a single Graph.

networkx.parse_sparse6

parse_sparse6(*str*)
> Read undirected graph in sparse6 format.

networkx.read_sparse6_list

read_sparse6_list(*path*)
> Read simple undirected graphs in sparse6 format from path. Returns a list of Graphs, one for each line in file.

3.8.9 Pajek

Read graphs in Pajek format.

See http://vlado.fmf.uni-lj.si/pub/networks/pajek/doc/draweps.htm for format information.

This implementation handles only directed and undirected graphs including those with self loops and parallel edges.

read_pajek(path)	Read graph in Pajek format from path.
write_pajek(G, path)	Write in Pajek format to path.
parse_pajek(lines[, edge_attr])	Parse pajek format graph from string or iterable.

networkx.read_pajek

read_pajek(*path*)
    Read graph in Pajek format from path.

    Returns a MultiGraph or MultiDiGraph.

> **Parameters path** : file or string
>
> > File or filename to write. Filenames ending in .gz or .bz2 will be compressed.

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)
```

networkx.write_pajek

write_pajek(*G, path*)
    Write 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.

Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_pajek(G, "test.net")
```

networkx.parse_pajek

parse_pajek(*lines, edge_attr=True*)

    Parse pajek format graph from string or iterable.

    Primarily used as a helper for read_pajek().

    See Also:

    read_pajek

3.9 Drawing

3.9.1 Matplotlib

Draw networks with matplotlib (pylab).

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, pos, **kwds[, with_labels])	Draw the graph G with given node positions pos
draw_networkx_nodes(G, pos, **kwds[, ...])	Draw nodes of 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

networkx.draw

draw(*G, pos=None, ax=None, hold=None, **kwds*)

    Draw the graph G with matplotlib (pylab).

    This is a pylab friendly function that will use the current pylab figure axes (e.g. subplot).

    pos is a dictionary keyed by vertex with a two-tuple of x-y positions as the value. See networkx.layout for functions that compute node positions.

    Usage:

```
>>> from networkx import *
>>> G=dodecahedral_graph()
>>> draw(G)
>>> pos=graphviz_layout(G)
>>> draw(G,pos)
>>> draw(G,pos=spring_layout(G))
```

    Also see doc/examples/draw_*

Parameters

- *nodelist*: list of nodes to be drawn (default=G.nodes())
- *edgelist*: list of edges to be drawn (default=G.edges())
- *node_size*: scalar or array of the same length as nodelist (default=300)
- *node_color*: single color string or numeric/numarray array of floats (default='r')
- *node_shape*: node shape (default='o'), or 'so^>v<dph8' see pylab.scatter
- *alpha*: transparency (default=1.0)
- *cmap*: colormap for mapping intensities (default=None)
- *vmin,vmax*: min and max for colormap scaling (default=None)
- *width*: line width of edges (default =1.0)
- *edge_color*: scalar or array (default='k')
- *edge_cmap*: colormap for edge intensities (default=None)
- *edge_vmin,edge_vmax*: min and max for colormap edge scaling (default=None)
- *style*: edge linestyle (default='solid') (solid|dashed|dotted,dashdot)
- *labels*: dictionary keyed by node of text labels (default=None)
- *font_size*: size for text labels (default=12)
- *font_color*: (default='k')
- *font_weight*: (default='normal')
- *font_family*: (default='sans-serif')
- *ax*: matplotlib axes instance

for more see pylab.scatter

NB: this has the same name as pylab.draw so beware when using

```
>>> from networkx import *
```

since you will overwrite the pylab.draw function.

A good alternative is to use

```
>>> import pylab as P
>>> import networkx as NX
>>> G=NX.dodecahedral_graph()
```

and then use

```
>>> NX.draw(G)   # networkx draw()
```

and >>> P.draw() # pylab draw()

networkx.draw_networkx

draw_networkx (*G, pos, with_labels=True, **kwds*)

Draw the graph G with given node positions pos

Usage:

```
>>> from networkx import *
>>> import pylab as P
>>> ax=P.subplot(111)
>>> G=dodecahedral_graph()
>>> pos=spring_layout(G)
>>> draw_networkx(G,pos,ax=ax)
```

This is same as 'draw' but the node positions *must* be specified in the variable pos. pos is a dictionary keyed by vertex with a two-tuple of x-y positions as the value. See networkx.layout for functions that compute node positions.

An optional matplotlib axis can be provided through the optional keyword ax.

with_labels contols text labeling of the nodes

Also see:

draw_networkx_nodes() draw_networkx_edges() draw_networkx_labels()

networkx.draw_networkx_nodes

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 nodes of graph G

This draws only the nodes of the graph G.

pos is a dictionary keyed by vertex with a two-tuple of x-y positions as the value. See networkx.layout for functions that compute node positions.

nodelist is an optional list of nodes in G to be drawn. If provided only the nodes in nodelist will be drawn.

see draw_networkx for the list of other optional parameters.

networkx.draw_networkx_edges

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.

pos is a dictionary keyed by vertex with a two-tuple of x-y positions as the value. See networkx.layout for functions that compute node positions.

edgelist is an optional list of the edges in G to be drawn. If provided, only the edges in edgelist will be drawn.

edgecolor can be a list of matplotlib color letters such as 'k' or 'b' that lists the color of each edge; the list must be ordered in the same way as the edge list. Alternatively, this list can contain numbers and those number are mapped to a color scale using the color map edge_cmap. Finally, it can also be a list of (r,g,b) or (r,g,b,a) tuples, in which case these will be used directly to color the edges. If the latter mode is used, you should not provide a value for alpha, as it would be applied globally to all lines.

For directed graphs, "arrows" (actually just thicker stubs) are drawn at the head end. Arrows can be turned off with keyword arrows=False.

See draw_networkx for the list of other optional parameters.

networkx.draw_networkx_labels

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

  pos is a dictionary keyed by vertex with a two-tuple of x-y positions as the value. See networkx.layout for functions that compute node positions.

  labels is an optional dictionary keyed by vertex with node labels as the values. If provided only labels for the keys in the dictionary are drawn.

  See draw_networkx for the list of other optional parameters.

networkx.draw_networkx_edge_labels

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, **kwds*)
  Draw edge labels.

  pos is a dictionary keyed by vertex with a two-tuple of x-y positions as the value. See networkx.layout for functions that compute node positions.

  labels is an optional dictionary keyed by edge tuple with labels as the values. If provided only labels for the keys in the dictionary are drawn. If not provided the edge data is used as a label.

  See draw_networkx for the list of other optional parameters.

networkx.draw_circular

draw_circular(*G, **kwargs*)
  Draw the graph G with a circular layout

networkx.draw_random

draw_random(*G, **kwargs*)
  Draw the graph G with a random layout.

networkx.draw_spectral

draw_spectral(*G, **kwargs*)
  Draw the graph G with a spectral layout.

networkx.draw_spring

draw_spring(*G, **kwargs*)
  Draw the graph G with a spring layout

networkx.draw_shell

draw_shell (*G, **kwargs*)
> Draw networkx graph with shell layout

networkx.draw_graphviz

draw_graphviz (*G, prog='neato', **kwargs*)
> Draw networkx graph with graphviz layout

3.9.2 Graphviz AGraph (dot)

Interface to pygraphviz AGraph class.

Examples

```
>>> G=nx.complete_graph(5)
>>> A=nx.to_agraph(G)
>>> H=nx.from_agraph(A)
```

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[, create_using])	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.

networkx.from_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)
```

networkx.to_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)
```

networkx.write_dot

write_dot(*G, path*)

Write NetworkX graph G to Graphviz dot format on path.

Parameters **G** : graph

A networkx graph

path : filename

Filename or file handle to write.

networkx.read_dot

read_dot(*path, create_using=None*)

Return a NetworkX graph from a dot file on path.

Parameters **path** : file or string

File name or file handle to read.

> **create_using** : Graph container, optional
>
> > Use specified Graph container to build graph. The default is nx.Graph().

networkx.graphviz_layout

graphviz_layout (*G, prog='neato', root=None, args=''*)

> Create node positions for G using Graphviz.
>
> > **Parameters** **G** : NetworkX graph
> >
> > > A graph created with NetworkX
> >
> > **prog** : string
> >
> > > Name of Graphviz layout program
> >
> > **root** : string, optional
> >
> > > Root node for twopi layout
> >
> > **args** : string, optional
> >
> > > Extra arguments to Graphviz layout program
> >
> > **Returns** : dictionary
> >
> > > 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')
```

networkx.pygraphviz_layout

pygraphviz_layout (*G, prog='neato', root=None, args=''*)

> Create node positions for G using Graphviz.
>
> > **Parameters** **G** : NetworkX graph
> >
> > > A graph created with NetworkX
> >
> > **prog** : string
> >
> > > Name of Graphviz layout program
> >
> > **root** : string, optional
> >
> > > Root node for twopi layout
> >
> > **args** : string, optional
> >
> > > Extra arguments to Graphviz layout program

> **Returns** : 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')
```

3.9.3 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.

See Also

Pydot: http://www.dkbza.org/pydot.html Graphviz: http://www.research.att.com/sw/tools/graphviz/ DOT Language: http://www.research.att.com/~erg/graphviz/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[, create_using])	Return a NetworkX graph from a dot file on path.
graphviz_layout(G[, prog, root, args])	Create node positions for G using Graphviz.
pydot_layout(G, **kwds[, prog, root])	Create node positions using Pydot and Graphviz.

networkx.from_pydot

from_pydot (*P*)

> Return a NetworkX graph from a Pydot graph.
>
> > **Parameters** **P** : Pydot graph
> >
> > > A graph created with Pydot

Examples

```
>>> K5=nx.complete_graph(5)
>>> A=nx.to_pydot(K5)
>>> G=nx.from_pydot(A)
```

networkx.to_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)
```

networkx.write_dot

write_dot (*G, path*)

Write NetworkX graph G to Graphviz dot format on path.

> **Parameters** **G** : graph
>
> > A networkx graph
>
> **path** : filename
>
> > Filename or file handle to write.

networkx.read_dot

read_dot (*path, create_using=None*)

Return a NetworkX graph from a dot file on path.

> **Parameters** **path** : file or string
>
> > File name or file handle to read.
>
> **create_using** : Graph container, optional
>
> > Use specified Graph container to build graph. The default is nx.Graph().

networkx.graphviz_layout

graphviz_layout (*G, prog='neato', root=None, args=''*)

Create node positions for G using Graphviz.

> **Parameters** **G** : NetworkX graph
>
> > A graph created with NetworkX
>
> **prog** : string
>
> > Name of Graphviz layout program
>
> **root** : string, optional
>
> > Root node for twopi layout
>
> **args** : string, optional
>
> > Extra arguments to Graphviz layout program
>
> **Returns** : dictionary
>
> > 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')
```

networkx.pydot_layout

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')
```

3.9.4 Graph Layout

Node positioning algorithms for graph drawing.

circular_layout(G[, dim, scale])	Position nodes on a circle.
random_layout(G[, dim])	
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.

networkx.circular_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)
```

networkx.random_layout

random_layout(*G, dim=2*)

networkx.shell_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.circular_layout(G,shells)
```

networkx.spring_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)

networkx.spectral_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 unidrected 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)
```

3.10 Exceptions

Base exceptions and errors for NetworkX.

class **NetworkXException**()
     Base class for exceptions in NetworkX.

class **NetworkXError**()
     Exception for a serious error in NetworkX

3.11 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
```

3.11.1 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.

networkx.utils.is_string_like

is_string_like(*obj*)
     Check if obj is string.

networkx.utils.flatten

flatten(*obj, result=None*)
     Return flattened version of (possibly nested) iterable object.

networkx.utils.iterable

iterable(*obj*)
> Return True if obj is iterable with a well-defined len()

networkx.utils.is_list_of_ints

is_list_of_ints(*intlist*)
> Return True if list is a list of ints.

networkx.utils._get_fh

_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'.

3.11.2 Data structures and Algorithms

UnionFind.union(*objects)	Find the sets containing the objects and merge them all.

networkx.utils.UnionFind.union

union(**objects*)
> Find the sets containing the objects and merge them all.

3.11.3 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.

networkx.utils.pareto_sequence

pareto_sequence(*n, exponent=1.0*)
> Return sample sequence of length n from a Pareto distribution.

networkx.utils.powerlaw_sequence

powerlaw_sequence(*n, exponent=2.0*)
> Return sample sequence of length n from a power law distribution.

networkx.utils.uniform_sequence

uniform_sequence(*n*)

 Return sample sequence of length n from a uniform distribution.

networkx.utils.cumulative_distribution

cumulative_distribution(*distribution*)

 Return normalized cumulative distribution from discrete distribution.

networkx.utils.discrete_sequence

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

3.11.4 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

networkx.utils.scipy_pareto_sequence

scipy_pareto_sequence(*n, exponent=1.0*)

 Return sample sequence of length n from a Pareto distribution.

networkx.utils.scipy_powerlaw_sequence

scipy_powerlaw_sequence(*n, exponent=2.0*)

 Return sample sequence of length n from a power law distribution.

networkx.utils.scipy_poisson_sequence

scipy_poisson_sequence(*n, mu=1.0*)

 Return sample sequence of length n from a Poisson distribution.

networkx.utils.scipy_uniform_sequence

scipy_uniform_sequence(*n*)
    Return sample sequence of length n from a uniform distribution.

networkx.utils.scipy_discrete_sequence

scipy_discrete_sequence(*n, distribution=False*)
    Return sample sequence of length n from a given discrete distribution

    distribution=histogram of values, will be normalized

3.12 Glossary

edge FIXME

hashable FIXME

nbunch FIXME

node FIXME

HISTORY

Original Creators:

```
Aric Hagberg, hagberg@lanl.gov
Pieter Swart, swart@lanl.gov
Dan Schult, dschult@colgate.edu
```

4.1 API changes

4.1.1 Version 1.0 notes and API changes

We have made some significant API changes, detailed below, to add functionality and clarity. This page reflects changes from networkx-0.99 to networkx-1.0. For changes from earlier versions to networkx-0.99 see `Version 0.99 API changes`.

Version 1.0 requires Python 2.4 or greater.

Please send comments and questions to the networkx-discuss mailing list: http://groups.google.com/group/networkx-discuss .

Version numbering

In the future we will use a more standard release numbering system with major.minor[build] labels where major and minor are numbers and [build] is a label such as "dev1379" to indicate a development version or "rc1" to indicate a release candidate.

We plan on sticking closer to a time-based release schedule with smaller incremental changes released on a roughly quarterly basis. The graph classes API will remain fixed, unless we determine there are serious bugs or other defects in the existing classes, until networkx-2.0 is released at some time in the future.

Changes in base classes

The most significant changes in are in the graph classes. All of the graph classes now allow optional graph, node, and edge attributes. Those attributes are stored internally in the graph classes as dictionaries and can be accessed simply like Python dictionaries in most cases.

Graph attributes

Each graph keeps a dictionary of key=value attributes in the member G.graph. These attributes can be accessed directly using G.graph or added at instantiation using keyword arguments.

```
>>> G=nx.Graph(region='Africa')
>>> G.graph['color']='green'
>>> G.graph
{'region': 'Africa', 'color': 'green'}
```

Node attributes

Each node has a corresponding dictionary of attributes. Adding attributes to nodes is optional.

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'})]
```

Edge attributes

Each edge has a corresponding dictionary of attributes. The default edge data is now an empty dictionary of attributes and adding attributes to edges is optional.

A common use case is to add a weight attribute to an edge:

```
>>> G.add_edge(1,2,weight=3.14159)
```

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
```

Methods changed

Graph(), DiGraph(), MultiGraph(), MultiDiGraph()

> Now takes optional keyword=value attributes on initialization.
>
> ```
> >>> G=Graph(year='2009',city='New York')
> ```

---

add_node()

Now takes optional keyword=value attributes or a dictionary of attributes.

```
>>> G.add_node(1,room=714)
```

add_nodes_from()

Now takes optional keyword=value attributes or a dictionary of attributes applied to all affected nodes.

```
>>> G.add_nodes_from([1,2],time='2pm')  # all nodes have same attribute
```

add_edge()

Now takes optional keyword=value attributes or a dictionary of attributes.

```
>>> G.add_edge(1, 2, weight=4.7 )
```

add_edges_from()

Now takes optional keyword=value attributes or a dictionary of attributes applied to all affected edges.

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

nodes() and nodes_iter()

New keyword data=True|False keyword determines whether to return two-tuples (n,dict) (True) with node attribution dictionary

```
>>> G=Graph([(1,2),(3,4)]
>>> G.nodes(data=True)
[(1, {}), (2, {}), (3, {}), (4, {})]
```

copy()

Now returns a deep copy of the graph (copies all underlying data and attributes for nodes and edges). Use the class initializer to make a shallow copy:

```
>>> G=Graph()
>>> G_shallow=Graph(G) # shallow copy
>>> G_deep=G.copy() # deep copy
```

to_directed(), to_undirected()

Now returns a deep copy of the graph (copies all underlying data and attributes for nodes and edges). Use the class initializer to make a shallow copy:

```
>>> G=Graph()
>>> D_shallow=DiGraph(G) # shallow copy
>>> D_deep=G.to_directed() # deep copy
```

subgraph()

> With copy=True now returns a deep copy of the graph (copies all underlying data and attributes for nodes
> and edges).

```
>>> G=Graph()
>>> H=G.subgraph(copy=True) # deep copy of all data
```

add_cycle(), add_path(), add_star()

> Now take optional keyword=value attributes or a dictionary of attributes which are applied to all edges
> affected by the method.

```
>>> G=Graph()
>>> G.add_path([0,1,2,3],width=3.2)
```

Methods removed

delete_node()

> The preferred name is now remove_node().

delete_nodes_from()

> No longer raises an exception on an attempt to delete a node not in the graph. The preferred name is now
> remove_nodes_from().

delete_edge()

> Now raises an exception on an attempt to delete an edge not in the graph. The preferred name is now
> remove_edge().

delete_edges_from()

> The preferred name is now remove_edges_from().

has_neighbor():

> Use has_edge()

get_edge()

> Renamed to get_edge_data(). Returns the edge attribute dictionary.

> The fastest way to get edge data for edge (u,v) is to use G[u][v] instead of G.get_edge_data(u,v)

Members removed

directed, multigraph, weighted

> Use methods G.is_directed() and G.is_multigraph(). All graphs are weighted graphs now if they have
> numeric values in the 'weight' edge attribute.

Methods added

add_weighted edges_from()

> Convenience method to add weighted edges to graph using a list of 3-tuples (u,v,weight).

get_edge_data()

> Renamed from get_edge().

> The fastest way to get edge data for edge (u,v) is to use G[u][v] instead of G.get_edge_data(u,v)

is_directed()

> replaces member G.directed

is_multigraph()

> replaces member G.multigraph

Classes Removed

LabeledGraph, LabeledDiGraph

> These classes have been folded into the regular classes.

UbiGraph

> Removed as the ubigraph platform is no longer being supported.

Additional functions/generators

ego_graph, stochastic_graph, PageRank algorithm, HITS algorithm, GraphML writer, freeze, is_frozen, A* algorithm, directed scale-free generator, random clustered graph.

Converting your existing code to networkx-1.0

Weighted edges

Edge information is now stored in an attribution dictionary so all edge data must be given a key to identify it.

There is currently only one standard/reserved key, 'weight', which is used by algorithms and functions that use weighted edges. The associated value should be numeric. All other keys are available for users to assign as needed.

```
>>> G.add_edge(1,2,weight=3.1415) # add the edge 1-2 with a weight
>>> G[1][2]['weight']=2.3 # set the weight to 2.3
```

Similarly, for direct access the edge data, use the key of the edge data to retrieve it.

```
>>> w = G[1][2]['weight']
```

All NetworkX algorithms that require/use weighted edges now use the 'weight' edge attribute. If you have existing algorithms that assumed the edge data was numeric, you should replace G[u][v] and G.get_edge(u,v) with G[u][v]['weight'].

An idiom for getting a weight for graphs with or without an assigned weight key is

```
>>> w= G[u][v].get('weight',1)  # set w to 1 if there is no 'weight' key
```

4.1.2 Version 0.99 API changes

The version networkx-0.99 is the penultimate release before networkx-1.0. We have bumped the version from 0.37 to 0.99 to indicate (in our unusual version number scheme) that this is a major change to NetworkX.

We have made some significant changes, detailed below, to NetworkX to improve performance, functionality, and clarity.

Version 0.99 requires Python 2.4 or greater.

Please send comments and questions to the networkx-discuss mailing list. http://groups.google.com/group/networkx-discuss

Changes in base classes

The most significant changes are in the graph classes. We have redesigned the Graph() and DiGraph() classes to optionally allow edge data. This change allows Graph and DiGraph to naturally represent weighted graphs and to hold arbitrary information on edges.

- Both Graph and DiGraph take an optional argument weighted=True|False. When weighted=True the graph is assumed to have numeric edge data (with default 1). The Graph and DiGraph classes in earlier versions used the Python None as data (which is still allowed as edge data).

- The Graph and DiGraph classes now allow self loops.

- The XGraph and XDiGraph classes are removed and replaced with MultiGraph and MultiDiGraph. MultiGraph and MultiDiGraph optionally allow parallel (multiple) edges between two nodes.

The mapping from old to new classes is as follows:

```
- Graph -> Graph (self loops allowed now, default edge data is 1)
- DiGraph -> DiGraph (self loops allowed now, default edge data is 1)
- XGraph(multiedges=False) -> Graph
- XGraph(multiedges=True) -> MultiGraph
- XDiGraph(multiedges=False) -> DiGraph
- XDiGraph(multiedges=True) -> MultiDiGraph
```

Methods changed

edges()

> New keyword data=True|False keyword determines whether to return two-tuples (u,v) (False) or three-tuples (u,v,d) (True)

delete_node()

> The preferred name is now remove_node().

delete_nodes_from()

No longer raises an exception on an attempt to delete a node not in the graph. The preferred name is now remove_nodes_from().

delete_edge()

Now raises an exception on an attempt to delete an edge not in the graph. The preferred name is now remove_edge().

delete_edges_from()

The preferred name is now remove_edges_from().

add_edge()

The add_edge() method no longer accepts an edge tuple (u,v) directly. The tuple must be unpacked into individual nodes.

```
>>> import networkx as nx
>>> u='a'
>>> v='b'
>>> e=(u,v)
>>> G=nx.Graph()
```

Old

```
>>> # G.add_edge((u,v))   # or G.add_edge(e)
```

New

```
>>> G.add_edge(*e) # or G.add_edge(*(u,v))
```

The * operator unpacks the edge tuple in the argument list.

Add edge now has a data keyword parameter for setting the default (data=1) edge data.

```
>>> G.add_edge('a','b','foo')  # add edge with string "foo" as data
>>> G.add_edge(1,2,5.0)  # add edge with float 5 as data
```

add_edges_from()

Now can take list or iterator of either 2-tuples (u,v), 3-tuples (u,v,data) or a mix of both.

Now has data keyword parameter (default 1) for setting the edge data for any edge in the edge list that is a 2-tuple.

has_edge()

The has_edge() method no longer accepts an edge tuple (u,v) directly. The tuple must be unpacked into individual nodes.

Old:

```
>>> # G.has_edge((u,v))   # or has_edge(e)
```

New:

```
>>> G.has_edge(*e) # or has_edge(*(u,v))
True
```

The * operator unpacks the edge tuple in the argument list.

get_edge()

Now has the keyword argument "default" to specify what value to return if no edge is found. If not specified an exception is raised if no edge is found.

The fastest way to get edge data for edge (u,v) is to use G[u][v] instead of G.get_edge(u,v)

degree_iter()

The degree_iter method now returns an iterator over pairs of (node, degree). This was the previous behavior of degree_iter(with_labels=true) Also there is a new keyword weighted=False|True for weighted degree.

subgraph()

The argument inplace=False|True has been replaced with copy=True|False.

Subgraph no longer takes create_using keyword. To change the graph type either make a copy of the graph first and then change type or change type and make a subgraph. E.g.

```
>>> G=nx.path_graph(5)
>>> H=nx.DiGraph(G.subgraph([0,1])) # digraph of copy of induced subgraph
```

__getitem__()

Getting node neighbors from the graph with G[v] now returns a dictionary.

```
>>> G=nx.path_graph(5)
>>>  G[0]
{1: 1}
```

To get a list of neighbors you can either use the keys of that dictionary or use

```
>>> G.neighbors(0)
[1]
```

This change allows algorithms to use the underlying dict-of-dict representation through G[v] for substantial performance gains. Warning: The returned dictionary should not be modified as it may corrupt the graph data structure. Make a copy G[v].copy() if you wish to modify the dict.

Methods removed

info()

now a function

```
>>> G=nx.Graph(name='test me')
>>> nx.info(G)
Name:                test me
Type:                Graph
Number of nodes:  0
Number of edges:  0
```

node_boundary()

now a function

edge_boundary()

now a function

is_directed()

use the directed attribute

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

G.out_edges()

use G.edges()

G.in_edges()

use

```
>>> G=nx.DiGraph()
>>> R=G.reverse()
>>> R.edges()
[]
```

or

```
>>> [(v,u) for (u,v) in G.edges()]
[]
```

Methods added

adjacency_list()   Returns a list-of-lists adjacency list representation of the graph.

adjacency_iter()   Returns an iterator of (node, adjacency_dict[node]) over all nodes in the graph. Intended for fast access to the internal data structure for use in internal algorithms.

Other possible incompatibilities with existing code

Imports

Some of the code modules were moved into subdirectories.

Import statements such as:

```python
import networkx.centrality
from networkx.centrality import *
```

may no longer work (including that example).

Use either

```python
>>> import networkx # e.g. centrality functions available as networkx.fcn()
```

or

```python
>>> from networkx import * # e.g. centrality functions available as fcn()
```

Self-loops

For Graph and DiGraph self loops are now allowed. This might affect code or algorithms that add self loops which were intended to be ignored.

Use the methods

- nodes_with_selfloops()
- selfloop_edges()
- number_of_selfloops()

to discover any self loops.

Copy

Copies of NetworkX graphs including using the copy() method now return complete copies of the graph. This means that all connection information is copied–subsequent changes in the copy do not change the old graph. But node keys and edge data in the original and copy graphs are pointers to the same data.

prepare_nbunch

Used internally - now called nbunch_iter and returns an iterator.

Converting your old code to Version 0.99

Mostly you can just run the code and python will raise an exception for features that changed. Common places for changes are

- Converting XGraph() to either Graph or MultiGraph
- Converting XGraph.edges() to Graph.edges(data=True)

- Switching some rarely used methods to attributes (e.g. directed) or to functions (e.g. node_boundary)

- If you relied on the old default edge data being None, you will have to account for it now being 1.

You may also want to look through your code for places which could improve speed or readability. The iterators are helpful with large graphs and getting edge data via G[u][v] is quite fast. You may also want to change G.neighbors(n) to G[n] which returns the dict keyed by neighbor nodes to the edge data. It is faster for many purposes but does not work well when you are changing the graph.

4.2 Release Log

4.2.1 Networkx-1.0

Release date: TBD August 2009

See: https://networkx.lanl.gov/trac/timeline

New features

This release has sigificant changes to parts of the graph API to allow graph, node, and edge attributes. See http://networkx.lanl.gov//reference/api_changes.html

- Update Graph, DiGraph, and MultiGraph classes to allow attributes.

- Default edge data is now an empty dictionary (was the integer 1)

- Many more, see https://networkx.lanl.gov/trac/query?status=closed&group=milestone&milestone=networkx-1.0

Examples

- Update to work with networkx-1.0 API

4.2.2 Networkx-0.99

Release date: 18 November 2008

See: https://networkx.lanl.gov/trac/timeline

New features

This release has sigificant changes to parts of the graph API. See http://networkx.lanl.gov//reference/api_changes.html

- Update Graph and DiGraph classes to use weighted graphs as default Change in API for performance and code simplicity.

- New MultiGraph and MultiDiGraph classes (replace XGraph and XDiGraph)

- Update to use Sphinx documentation system http://networkx.lanl.gov/

- Developer site at https://networkx.lanl.gov/trac/

- Experimental LabeledGraph and LabeledDiGraph

- Moved package and file layout to subdirectories.

Bug fixes

- handle root= option to draw_graphviz correctly

Examples

- Update to work with networkx-0.99 API

- Drawing examples now use matplotlib.pyplot interface

- Improved drawings in many examples

- New examples - see http://networkx.lanl.gov/examples/

4.2.3 NetworkX-0.37

Release date: 17 August 2008

See: https://networkx.lanl.gov/trac/timeline

NetworkX now requires Python 2.4 or later for full functionality.

New features

- Edge coloring and node line widths with Matplotlib drawings

- Update pydot functions to work with pydot-1.0.2

- Maximum-weight matching algorithm

- Ubigraph interface for 3D OpenGL layout and drawing

- Pajek graph file format reader and writer

- p2g graph file format reader and writer

- Secondary sort in topological sort

Bug fixes

- Better edge data handling with GML writer

- Edge betweenness fix for XGraph with default data of None

- Handle Matplotlib version strings (allow "pre")

- Interface to PyGraphviz (to_agraph()) now handles parallel edges

- Fix bug in copy from XGraph to XGraph with multiedges

- Use SciPy sparse lil matrix format instead of coo format

- Clear up ambiguous cases for Barabasi-Albert model

- Better care of color maps with Matplotlib when drawing colored nodes and edges

- Fix error handling in layout.py

Examples

- Ubigraph examples showing 3D drawing

4.2.4 NetworkX-0.36

Release date: 13 January 2008

See: https://networkx.lanl.gov/trac/timeline

New features

- GML format graph reader, tests, and example (football.py)
- edge_betweenness() and load_betweenness()

Bug fixes

- remove obsolete parts of pygraphviz interface
- improve handling of Matplotlib version strings
- write_dot() now writes parallel edges and self loops
- is_bipartite() and bipartite_color() fixes
- configuration model speedup using random.shuffle()
- convert with specified nodelist now works correctly
- vf2 isomorphism checker updates

4.2.5 NetworkX-0.35.1

Release date: 27 July 2007

See: https://networkx.lanl.gov/trac/timeline

Small update to fix import readwrite problem and maintain Python2.3 compatibility.

4.2.6 NetworkX-0.35

Release date: 22 July 2007

See: https://networkx.lanl.gov/trac/timeline

New features

- algorithms for strongly connected components.
- Brandes betweenness centrality algorithm (weighted and unweighted versions)
- closeness centrality for weighted graphs
- dfs_preorder, dfs_postorder, dfs_tree, dfs_successor, dfs_predecessor
- readers for GraphML, LEDA, sparse6, and graph6 formats.

- allow arguments in graphviz_layout to be passed directly to graphviz

Bug fixes

- more detailed installation instructions
- replaced dfs_preorder,dfs_postorder (see search.py)
- allow initial node positions in spectral_layout
- report no error on attempting to draw empty graph
- report errors correctly when using tuples as nodes #114
- handle conversions from incomplete dict-of-dict data

4.2.7 NetworkX-0.34

Release date: 12 April 2007

See: https://networkx.lanl.gov/trac/timeline

New features

- benchmarks for graph classes
- Brandes betweenness centrality algorithm
- Dijkstra predecessor and distance algorithm
- xslt to convert DIA graphs to NetworkX
- number_of_edges(u,v) counts edges between nodes u and v
- run tests with python setup_egg.py test (needs setuptools) else use python -c "import networkx; networkx.test()"
- is_isomorphic() that uses vf2 algorithm

Bug fixes

- speedups of neighbors()
- simplified Dijkstra's algorithm code
- better exception handling for shortest paths
- get_edge(u,v) returns None (instead of exception) if no edge u-v
- floyd_warshall_array fixes for negative weights
- bad G467, docs, and unittest fixes for graph atlas
- don't put nans in numpy or scipy sparse adjacency matrix
- handle get_edge() exception (return None if no edge)
- remove extra kwds arguments in many places
- no multi counting edges in conversion to dict of lists for multigraphs
- allow passing tuple to get_edge()

- bad parameter order in node/edge betweenness

- edge betweenness doesn't fail with XGraph

- don't throw exceptions for nodes not in graph (silently ignore instead) in edges_* and degree_*

4.2.8 NetworkX-0.33

Release date: 27 November 2006

See: https://networkx.lanl.gov/trac/timeline

New features

- draw edges with specified colormap

- more efficient version of Floyd's algorithm for all pairs shortest path

- use numpy only, Numeric is deprecated

- include tests in source package (networkx/tests)

- include documentation in source package (doc)

- **tests can now be run with**

    ```
    >>> import networkx
    >>> networkx.test()
    ```

Bug fixes

- read_gpickle now works correctly with Windows

- refactored large modules into smaller code files

- degree(nbunch) now returns degrees in same order as nbunch

- degree() now works for multiedges=True

- update node_boundary and edge_boundary for efficiency

- edited documentation for graph classes, now mostly in info.py

Examples

- Draw edges with colormap

4.2.9 NetworkX-0.32

Release date: 29 September 2006

See: https://networkx.lanl.gov/trac/timeline

New features

- Update to work with numpy-1.0x

- Make egg usage optional: use python setup_egg.py bdist_egg to build egg

- Generators and functions for bipartite graphs

- Experimental classes for trees and forests

- Support for new pygraphviz update (in nx_agraph.py) , see http://networkx.lanl.gov/pygraphviz/ for pygraphviz details

Bug fixes

- Handle special cases correctly in triangles function

- Typos in documentation

- Handle special cases in shortest_path and shortest_path_length, allow cutoff parameter for maximum depth to search

- Update examples: erdos_renyi.py, miles.py, roget,py, eigenvalues.py

Examples

- Expected degree sequence

- New pygraphviz interface https://networkx.lanl.gov/trac/browser/networkx/trunk/doc/examples/pygraphviz_simple.py https://networkx.lanl.gov/trac/browser/networkx/trunk/doc/examples/pygraphviz_miles.py https://networkx.lanl.gov/trac/browser/networkx/trunk/doc/examples/pygraphviz_attributes.py

4.2.10 NetworkX-0.31

Release date: 20 July 2006

See: https://networkx.lanl.gov/trac/timeline

New features

- arbitrary node relabeling (use relabel_nodes)

- conversion of NetworkX graphs to/from Python dict/list types, numpy matrix or array types, and scipy_sparse_matrix types

- generator for random graphs with given expected degree sequence

Bug fixes

- Allow drawing graphs with no edges using pylab

- Use faster heapq in dijkstra

- Don't complain if X windows is not available

Examples

- update drawing examples

4.2.11 NetworkX-0.30

Release date: 23 June 2006

See: https://networkx.lanl.gov/trac/timeline

New features

- update to work with Python 2.5
- bidirectional version of shortest_path and Dijkstra
- single_source_shortest_path and all_pairs_shortest_path
- s-metric and experimental code to generate maximal s-metric graph
- double_edge_swap and connected_double_edge_swap
- Floyd's algorithm for all pairs shortest path
- read and write unicode graph data to text files
- read and write YAML format text files, http://yaml.org

Bug fixes

- speed improvements (faster version of subgraph, is_connected)
- added cumulative distribution and modified discrete distribution utilities
- report error if DiGraphs are sent to connected_components routines
- removed with_labels keywords for many functions where it was causing confusion
- function name changes in shortest_path routines
- saner internal handling of nbunch (node bunches), raise an exception if an nbunch isn't a node or iterable
- better keyword handling in io.py allows reading multiple graphs
- don't mix Numeric and numpy arrays in graph layouts and drawing
- avoid automatically rescaling matplotlib axes when redrawing graph layout

Examples

- unicode node labels

4.2.12 NetworkX-0.29

Release date: 28 April 2006

See: https://networkx.lanl.gov/trac/timeline

New features

- Algorithms for betweeenness, eigenvalues, eigenvectors, and spectral projection for threshold graphs
- Use numpy when available
- dense_gnm_random_graph generator
- Generators for some directed graphs: GN, GNR, and GNC by Krapivsky and Redner
- Grid graph generators now label by index tuples. Helper functions for manipulating labels.
- relabel_nodes_with_function

Bug fixes

- Betweenness centrality now correctly uses Brandes definition and has normalization option outside main loop
- Empty graph now labled as empty_graph(n)
- shortest_path_length used python2.4 generator feature
- degree_sequence_tree off by one error caused nonconsecutive labeling
- periodic_grid_2d_graph removed in favor of grid_2d_graph with periodic=True

4.2.13 NetworkX-0.28

Release date: 13 March 2006

See: https://networkx.lanl.gov/trac/timeline

New features

- Option to construct Laplacian with rows and columns in specified order
- Option in convert_node_labels_to_integers to use sorted order
- predecessor(G,n) function that returns dictionary of nodes with predecessors from breadth-first search of G starting at node n. https://networkx.lanl.gov/trac/ticket/26

Examples

- Formation of giant component in binomial_graph:
- Chess masters matches:
- Gallery https://networkx.lanl.gov/gallery.html

Bug fixes

- **Adjusted names for random graphs.**
    - erdos_renyi_graph=binomial_graph=gnp_graph: n nodes with edge probability p
    - gnm_graph: n nodes and m edges
    - fast_gnp_random_graph: gnp for sparse graphs (small p)

- Documentation contains correct spelling of Barabási, Bollobás, Erdős, and Rényi in UTF-8 encoding

- Increased speed of connected_components and related functions by using faster BFS algorithm in networkx.paths https://networkx.lanl.gov/trac/ticket/27

- XGraph and XDiGraph with multiedges=True produced error on delete_edge

- Cleaned up docstring errors

- Normalize names of some graphs to produce strings that represent calling sequence

4.2.14 NetworkX-0.27

Release date: 5 February 2006

See: https://networkx.lanl.gov/trac/timeline

New features

- sparse_binomial_graph: faster graph generator for sparse random graphs

- read/write routines in io.py now handle XGraph() type and gzip and bzip2 files

- optional mapping of type for read/write routine to allow on-the-fly conversion of node and edge datatype on read

- Substantial changes related to digraphs and definitions of neighbors() and edges(). For digraphs edges=out_edges. Neighbors now returns a list of neighboring nodes with possible duplicates for graphs with parallel edges See https://networkx.lanl.gov/trac/ticket/24

- Addition of out_edges, in_edges and corresponding out_neighbors and in_neighbors for digraphs. For digraphs edges=out_edges.

Examples

- Minard's data for Napoleon's Russian campaign

Bug fixes

- XGraph(multiedges=True) returns a copy of the list of edges for get_edge()

4.2.15 NetworkX-0.26

Release date: 6 January 2006

New features

- Simpler interface to drawing with pylab

- G.info(node=None) function returns short information about graph or node

- adj_matrix now takes optional nodelist to force ordering of rows/columns in matrix

- optional pygraphviz and pydot interface to graphviz is now callable as "graphviz" with pygraphviz preferred. Use draw_graphviz(G).

Examples

- Several new examples showing how draw to graphs with various properties of nodes, edges, and labels

Bug fixes

- Default data type for all graphs is now None (was the integer 1)
- add_nodes_from now won't delete edges if nodes added already exist
- Added missing names to generated graphs
- Indexes for nodes in graphs start at zero by default (was 1)

4.2.16 NetworkX-0.25

Release date: 5 December 2005

New features

- Uses setuptools for installation http://peak.telecommunity.com/DevCenter/setuptools
- Improved testing infrastructure, can now run python setup.py test
- Added interface to draw graphs with pygraphviz https://networkx.lanl.gov/pygraphviz/
- is_directed() function call

Examples

- Email example shows how to use XDiGraph with Python objects as edge data

Documentation

- Reformat menu, minor changes to Readme, better stylesheet

Bug fixes

- use create_using= instead of result= keywords for graph types in all cases
- missing weights for degree 0 and 1 nodes in clustering
- configuration model now uses XGraph, returns graph with identical degree sequence as input sequence
- fixed dijstra priority queue
- fixed non-recursive toposort and is_directed_acyclic graph

4.2.17 NetworkX-0.24

Release date: 20 August 2005

Bug fixes

- Update of dijstra algorithm code

- dfs_successor now calls proper search method

- Changed to list compehension in DiGraph.reverse() for python2.3 compatibility

- Barabasi-Albert graph generator fixed

- Attempt to add self loop should add node even if parallel edges not allowed

4.2.18 NetworkX-0.23

Release date: 14 July 2005

The NetworkX web locations have changed:

http://networkx.lanl.gov/ - main documentation site http://networkx.lanl.gov/svn/ - subversion source code repository https://networkx.lanl.gov/trac/ - bug tracking and info

Important Change

The naming conventions in NetworkX have changed. The package name "NX" is now "networkx".

The suggested ways to import the NetworkX package are

- import networkx

- import networkx as NX

- from networkx import *

New features

- DiGraph reverse

- **Graph generators**

    - watts_strogatz_graph now does rewiring method

    - old watts_strogatz_graph->newman_watts_strogatz_graph

Examples

Documentation

- Changed to reflect NX-networkx change

- main site is now https://networkx.lanl.gov/

Bug fixes

- Fixed logic in io.py for reading DiGraphs.

- Path based centrality measures (betweenness, closeness) modified so they work on graphs that are not connected and produce the same result as if each connected component were considered separately.

4.2.19 NetworkX-0.22

Release date: 17 June 2005

New features

- Topological sort, testing for directed acyclic graphs (DAGs)

- Dikjstra's algorithm for shortest paths in weighted graphs

- Multidimensional layout with dim=n for drawing

- 3d rendering demonstration with vtk

- **Graph generators**

    – random_powerlaw_tree

    – dorogovtsev_goltsev_mendes_graph

Examples

- Kevin Bacon movie actor graph: Examples/kevin_bacon.py

- Compute eigenvalues of graph Laplacian: Examples/eigenvalues.py

- Atlas of small graphs: Examples/atlas.py

Documentation

- Rewrite of setup scripts to install documentation and tests in documentation directory specified

Bug fixes

- Handle calls to edges() with non-node, non-iterable items.

- truncated_tetrahedral_graph was just plain wrong

- Speedup of betweenness_centrality code

- bfs_path_length now returns correct lengths

- Catch error if target of search not in connected component of source

- Code cleanup to label internal functions with _name

- Changed import statement lines to always use "import NX" to protect name-spaces

- Other minor bug-fixes and testing added

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 all those who have 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 and bugs and typos in the graph generators

- 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 contributed the code for matching all the graph types.

- 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().

- 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.

LEGAL

6.1 License

```
Copyright (C) 2004,2005 by
Aric Hagberg <hagberg@lanl.gov>
Dan Schult <dschult@colgate.edu>
Pieter Swart <swart@lanl.gov>
All rights reserved, see GNU_LGPL for details.

NetworkX is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 2.1 of the License, or (at your option) any later version.

NetworkX is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with NetworkX; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
```

6.2 Notice

```
This software and ancillary information (herein called SOFTWARE )
called NetworkX is made available under the terms described here. The
SOFTWARE has been approved for release with associated LA-CC number
04-061.

Unless otherwise indicated, this SOFTWARE has been authored by an
employee or employees of the University of California, operator of the
Los Alamos National Laboratory under Contract No. W-7405-ENG-36 with
the U.S. Department of Energy. The U.S. Government has rights to use,
reproduce, and distribute this SOFTWARE. The public may copy,
distribute, prepare derivative works and publicly display this
SOFTWARE without charge, provided that this Notice and any statement
of authorship are reproduced on all copies. Neither the Government nor
the University makes any warranty, express or implied, or assumes any
liability or responsibility for the use of this SOFTWARE.
```

```
If SOFTWARE is modified to produce derivative works, such modified
SOFTWARE should be clearly marked, so as not to confuse it with the
version available from LANL.
```

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

DOWNLOAD

8.1 Source and Binary Releases

http://cheeseshop.python.org/pypi/networkx/

http://networkx.lanl.gov/download/networkx/

8.2 Subversion Source Code Repository

Anonymous

svn checkout http://networkx.lanl.gov/svn/networkx/trunk networkx

Authenticated

svn checkout https://networkx.lanl.gov/svn/networkx/trunk networkx

8.3 Documentation

PDF

http://networkx.lanl.gov/networkx.pdf

HTML in zip file

http://networkx.lanl.gov/networkx-documentation.zip

BIBLIOGRAPHY

[BA02] R. Albert and A.-L. Barabási, "Statistical mechanics of complex networks", Reviews of Modern Physics, 74, pp. 47-97, 2002. (Preprint available online at http://citeseer.ist.psu.edu/442178.html or http://arxiv.org/abs/cond-mat/0106096)

[Bollobas01] B. Bollobás, "Random Graphs", Second Edition, Cambridge University Press, 2001.

[BE05] U. Brandes and T. Erlebach, "Network Analysis: Methodological Foundations", Lecture Notes in Computer Science, Volume 3418, Springer-Verlag, 2005.

[Diestel97] R. Diestel, "Graph Theory", Springer-Verlag, 1997. (A free electronic version is available at http://www.math.uni-hamburg.de/home/diestel/books/graph.theory/download.html)

[DM03] S.N. Dorogovtsev and J.F.F. Mendes, "Evolution of Networks", Oxford University Press, 2003.

[Langtangen04] H.P. Langtangen, "Python Scripting for Computational Science.", Springer Verlag Series in Computational Science and Engineering, 2004.

[Martelli03] A. Martelli, "Python in a Nutshell", O'Reilly Media Inc, 2003. (A useful guide to the language is available at http://www.oreilly.com/catalog/pythonian/chapter/ch04.pdf)

[Newman03] M.E.J. Newman, "The Structure and Function of Complex Networks", SIAM Review, 45, pp. 167-256, 2003. (Available online at http://epubs.siam.org/sam-bin/dbq/article/42480)

[Sedgewick02] R. Sedgewick, "Algorithms in C: Parts 1-4: Fundamentals, Data Structure, Sorting, Searching", Addison Wesley Professional, 3rd ed., 2002.

[Sedgewick01] R. Sedgewick, "Algorithms in C, Part 5: Graph Algorithms", Addison Wesley Professional, 3rd ed., 2001.

[West01] D. B. West, "Introduction to Graph Theory", Prentice Hall, 2nd ed., 2001.

[R1] A. L. Barabási and R. Albert "Emergence of scaling in random networks", Science 286, pp 509-512, 1999.

[R2] M.E.J. Newman, "The structure and function of complex networks", SIAM REVIEW 45-2, pp 167-256, 2003.

MODULE INDEX

N

INDEX

Symbols

to_dict_of_lists() (in module networkx), 209
to_directed() (Graph method), 43
to_directed() (MultiGraph method), 96
to_edgelist() (in module networkx), 210
to_numpy_matrix() (in module networkx), 211
to_pydot() (in module networkx), 231
to_scipy_sparse_matrix() (in module networkx), 212
to_undirected() (DiGraph method), 70
to_undirected() (MultiDiGraph method), 124
topological_sort() (in module networkx), 161
topological_sort_recursive() (in module networkx), 162
transitivity() (in module networkx), 141
triangles() (in module networkx), 141
trivial_graph() (in module networkx), 182
truncated_cube_graph() (in module networkx), 186
truncated_tetrahedron_graph() (in module networkx), 186
tutte_graph() (in module networkx), 186

U

uniform_sequence() (in module networkx.utils), 238
union() (in module networkx), 128
union() (UnionFind method), 237

W

watts_strogatz_graph() (in module networkx), 190
wheel_graph() (in module networkx), 182
write_adjlist() (in module networkx), 214
write_dot() (in module networkx), 229, 232
write_edgelist() (in module networkx), 218
write_gml() (in module networkx), 219
write_gpickle() (in module networkx), 220
write_multiline_adjlist() (in module networkx), 216
write_pajek() (in module networkx), 223
write_yaml() (in module networkx), 222