NetworkX Documentation

Release 0.99

Aric Hagberg, Dan Schult, Pieter Swart

November 18, 2008

CONTENTS

CHAPTER

ONE

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/

CHAPTER

TWO

TUTORIAL

2.1 Introduction

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**. We often will shorten NetworkX to "nx" in code examples by using the Python import

```
>>> import networkx as 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.

2.2 A Quick Tour

2.2.1 Building and drawing a small graph

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 all the examples below we assume that NX has been imported this way.

You may also use the usual mode for interactive experimentation that might clobber some names already in your name-space

```
>>> from networkx import *
```

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.

2.2.2 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(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.

2.2.3 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.has_neighbor(n1,n2)
- G.edges(), G.edges(n), G.edges(nbunch)
- G.edges_iter(), G.edges_iter(n), G.edges_iter(nbunch)
- G.get_edge(n1,n2)  # the object associated with this edge
- 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.)

2.3 Examples

Create an empty graph with no nodes and no edges.

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

G can be grown in several ways.

By adding one node at a time,

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

by adding a list of nodes,

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

or by adding any nbunch of nodes (see above definition of an nbunch),

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

(H can be a graph, iterator, string, set, or even a file.)

Any hashable object (except None) can represent a node, e.g. a text string, an image, an XML object, another Graph, a customized node object, etc.

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

(You should not change the object if the hash depends on its contents.)

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

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

or

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

or by adding a list of edges,

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

or by adding any ebunch of edges (see above definition of an ebunch),

```
>>> 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()
>>> 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'
```

will add new nodes/edges as required and stay quiet if they are already present.

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

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

We can examine them with

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

Removing nodes is similar:

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

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

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

Edge data/weights/labels/objects can also be associated with an edge:

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

Arbitrary objects can be associated with an edge. The 3-tuples (n1,n2,x) represent an edge between nodes n1 and n2 that is decorated with the object x (not necessarily hashable). For example, n1 and n2 can be protein objects from the RCSB Protein Data Bank, and x can refer to an XML record of a publication detailing experimental observations of their interaction.

You can see that nodes and edges are not NetworkX classes. 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,x).

2.3.1 Drawing a small graph

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 the drawing section for details.

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

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

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 http://matplotlib.sourceforge.net/faq/installing_faq.html#ma
compiled-fine-but-nothing-shows-up-with-plot

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.

Or to save drawings to a file, use, for example

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

to write 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')
```

2.3.2 Functions for analyzing graph properties

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]
```

Some functions defined on the nodes, e.g. degree() and clustering(), can be given a single node or an nbunch of nodes as argument. If a single node is specified, then a single value is returned. If an iterable nbunch is specified, then the function will return a list of values. With no argument, the function will return a list of values at all nodes of the graph.

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

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

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

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

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

2.3.3 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)
```

2.4 Graph IO

NetworkX can read and write graphs in many formats. See http://networkx.lanl.gov/reference/readwrite.html for a complete list of currently supported formats.

2.4.1 Reading a graph from a file

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

Write to adjacency list format

```
>>> nx.write_adjlist(G, "tetrahedral.adjlist")
```

Read from adjacency list format

```
>>> H=nx.read_adjlist("tetrahedral.adjlist")
```

Write to edge list format

```
>>> nx.write_edgelist(G, "tetrahedral.edgelist")
```

Read from edge list format

```
>>> H=nx.read_edgelist("tetrahedral.edgelist")
```

See also Interfacing with other tools below for how to draw graphs with matplotlib or graphviz.

2.5 Graphs with multiple edges

See the MultiGraph and MultiDiGraph classes. For example, to build Euler's famous graph of the bridges of Königsberg over the Pregel river, one can use

```
>>> K=nx.MultiGraph(name="Königsberg")
>>> K.add_edges_from([("A","B","Honey Bridge"),
...                   ("A","B","Blacksmith's Bridge"),
...                   ("A","C","Green Bridge"),
...                   ("A","C","Connecting Bridge"),
...                   ("A","D","Merchant's Bridge"),
...                   ("C","D","High Bridge"),
...                   ("B","D","Wooden Bridge")])
>>> K.degree("A")
5
```

2.6 Directed Graphs

The DiGraph class provides operations common to digraphs (graphs with directed edges). A subclass of Graph, Digraph adds the following methods to those of Graph:

- out_edges

- out_edges_iter
- in_edges
- in_edges_iter
- has_successor=has_neighbor
- has_predecessor
- successors=neighbors
- successors_iter=neighbors_iter
- predecessors
- predecessors_iter
- out_degree
- out_degree_iter
- in_degree
- in_degree_iter
- reverse

See networkx.DiGraph for more documentation.

2.7 Interfacing with other tools

NetworkX provides interfaces to Matplotlib and Graphviz for graph layout (node and edge positioning) and drawing. We also use matplotlib for graph spectra and in some drawing operations. Without either, one can still use the basic graph-related functions.

See the graph drawing section for details on how to install and use these tools.

2.7.1 Matplotlib

```
>>> G=nx.tetrahedral_graph()
>>> nx.draw(G)
```

2.7.2 Graphviz

```
>>> G=nx.tetrahedral_graph()
>>> nx.write_dot(G,"tetrahedral.dot")
```

2.8 Specialized Topics

2.8.1 Graphs composed of general objects

For most applications, nodes will have string or integer labels. The power of Python ("everything is an object") allows us to construct graphs with ANY hashable object as a node. (The Python object None is not allowed as a node). Note however that this will not work with non-Python datastructures, e.g. building a graph on a wrapped Python version of graphviz).

For example, one can construct a graph with Python mathematical functions as nodes, and where two mathematical functions are connected if they are in the same chapter in some Handbook of Mathematical Functions. E.g.

```
>>> from math import *
>>> G=nx.Graph()
>>> G.add_node(acos)
>>> G.add_node(sinh)
>>> G.add_node(cos)
>>> G.add_node(tanh)
>>> G.add_edge(acos,cos)
>>> G.add_edge(sinh,tanh)
>>> sorted(G.nodes())
[<built-in function acos>, <built-in function cos>, <built-in function sinh>, <built-in function tanh
```

As another example, one can build (meta) graphs using other graphs as the nodes.

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 convert_node_labels_to_integers() to obtain a more traditional graph with integer labels.

2.9 References

CHAPTER

THREE

REFERENCE

Release 0.99

Date November 18, 2008

3.1 Overview

NetworkX is built to allow easy creation, manipulation and measurement on large graph theoretic structures which we call networks. The graph theory literature defines a graph as a set of nodes(vertices) and edges(links) where each edge is associated with two nodes. In practical settings there are often properties associated with each node or edge. These structures are fully captured by NetworkX in a flexible and easy to learn environment.

3.1.1 Graphs

The basic object in NetworkX is a Graph. A simple type of Graph has undirected edges and does not allow multiple edges between nodes. The NetworkX Graph class contains this data structure. Other classes allow directed graphs (DiGraph) and multiple edges (MultiGraph/MultiDiGraph). All these classes allow nodes and edges to be associated with objects or data.

We will discuss the Graph class first.

Empty graphs are created by default.

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

At this point, the graph G has no nodes or edges and has the name "". The name can be set through the variable G.name or through an optional argument such as

```
>>> G=nx.Graph(name="I have a name!")
>>> print G.name
I have a name!
```

3.1.2 Graph Manipulation

Basic graph manipulation is provided through methods to add or remove nodes or edges.

```
>>> G.add_node(1)
>>> G.add_nodes_from(["red","blue","green"])
>>> G.remove_node(1)
>>> G.remove_nodes_from(["red","blue","green"])
>>> G.add_edge(1,2)
>>> G.add_edges_from([(1,3),(1,4)])
>>> G.remove_edge(1,2)
>>> G.remove_edges_from([(1,3),(1,4)])
```

If a node is removed, all edges associated with that node are also removed. If an edge is added connecting a node that does not yet exist, that node is added when the edge is added.

More complex manipulation is available through the methods:

```
G.add_star([list of nodes])   - add edges from the first node to all other nodes.
G.add_path([list of nodes])   - add edges to make this ordered path.
G.add_cycle([list of nodes])  - same as path, but connect ends.
G.subgraph([list of nodes]) - the subgraph of G containing those nodes.
G.clear()   - remove all nodes and edges.
G.copy()    - a copy of the graph.
G.to_undirected()  - return an undirected representation of G
G.to_directed()    - return a directed representation of G
```

Note: For G.copy() the graph structure is copied, but the nodes and edge data point to the same objects in the new and old graph.

In addition, the following functions are available:

```
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
```

You should also look at the graph generation routines in networkx.generators

3.1.3 Methods

Some properties are tied more closely to the data structure and can be obtained through methods as well as functions. These are:

```
- G.number_of_nodes()
- G.number_of_edges()
- G.nodes()          - a list of nodes in G.
- G.nodes_iter()     - an iterator over the nodes of G.
- G.has_node(v)      - check if node v in G (returns True or False).
- G.edges()          - a list of 2-tuples specifying edges of G.
- G.edges_iter()     - an iterator over the edges (as 2-tuples) of G.
- G.has_edge(u,v)    - check if edge (u,v) in G (returns True or False).
- G.neighbors(v)     - list of the neighbors of node v (outgoing if directed).
- G.neighbors_iter(v)      - an iterator over the neighbors of node v.
- G.degree()         - list of the degree of each node.
- G.degree_iter()    - an iterator yielding 2-tuples (node, degree).
```

Argument syntax and options are the same as for the functional form.

3.1.4 Node/Edge Reporting Methods

In many cases it is more efficient to loop using an iterator directly rather than creating a list. NX 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. The suffix "_iter" in a method name signifies that an iterator will be returned.

For node properties, an optional argument "with_labels" signifies whether the returned values should be identified by node or not. If "with_labels=False", only property values are returned. If "with_labels=True", a dict keyed by node to the property values is returned.

The following examples use the "triangles" function which returns the number of triangles a given node belongs to.

Example usage

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

3.1.5 Properties for specific nodes

Many node property functions return property values for either a single node, a list of nodes, or the whole graph. The return type is determined by an optional input argument.

1. By default, values are returned for all nodes in the graph.
2. If input is a list of nodes, a list of values for those nodes is returned.
3. If input is a single node, the value for that node is returned.

Node v is special for some reason. We want to print info on it.

```
>>> v=1
>>> print "Node %s has %s triangles."%(v,nx.triangles(G,v))
Node 1 has 3 triangles.
```

Maybe you need a polynomial on t?

```
>>> t=nx.triangles(G,v)
>>> poly=t**3+2*t-t+5
```

Get triangles for a subset of all nodes.

```
>>> vlist=range(0,4)
>>> triangle_dict = nx.triangles(G,vlist,with_labels=True)
>>> for (v,t) in triangle_dict.items():
...     print "Node %s is part of %s triangles."%(v,t)
Node 0 is part of 3 triangles.
Node 1 is part of 3 triangles.
Node 2 is part of 3 triangles.
Node 3 is part of 3 triangles.
```

3.2 Graph classes

3.2.1 Basic Graphs

The Graph and DiGraph classes provide simple graphs which allow self-loops. The default Graph and DiGraph are weighted graphs with edge weight equal to 1 but arbitrary edge data can be assigned.

Graph

Overview

class Graph (*data=None, name='', weighted=True*)

An undirected graph class without multiple (parallel) edges.

Nodes can be arbitrary (hashable) objects.

Arbitrary data/labels/objects can be associated with edges. The default data is 1. See add_edge and add_edges_from methods for details. Many NetworkX routines developed for weighted graphs assume this data is a number. Feel free to put other objects on the edges, but be aware that these weighted graph algorithms may give unpredictable results if the graph isn't a weighted graph.

Self loops are allowed.

Examples

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

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

G can be grown in several ways. By adding one node at a time:

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

by adding a list of nodes:

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

by using an iterator:

```
>>> G.add_nodes_from(xrange(100,110))
```

or by adding any container of nodes (a list, dict, set or even a file or the nodes from another graph).

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

Any hashable object (except None) can represent a node, e.g. a customized node object, or even another Graph.

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

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

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

by adding a list of edges:

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

or by adding any collection of edges:

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

Nodes will be added as needed when you add edges and there are no complaints when adding existing nodes or edges.

The default edge data is the number 1. To add edge information with an edge, use a 3-tuple (u,v,d).

```
>>> G.add_edges_from([(1,2,'blue'),(2,3,3.1)])
>>> G.add_edges_from( [(3,4),(4,5)], data='red')
>>> G.add_edge( 1, 2, 4.7 )
```

Adding and Removing Nodes and Edges

`Graph.add_node` (n)	Add a single node n.
`Graph.add_nodes_from` (nbunch)	Add nodes from nbunch.
`Graph.remove_node` (n)	Remove node n.
`Graph.remove_nodes_from` (nbunch)	Remove nodes specified in nbunch.
`Graph.add_edge` (u, v[, data])	Add an edge between u and v with optional data.
`Graph.add_edges_from` (ebunch[, data])	Add all the edges in ebunch.
`Graph.remove_edge` (u, v)	Remove the edge between (u,v).
`Graph.remove_edges_from` (ebunch)	Remove all edges specified in ebunch.
`Graph.add_star` (nlist[, data])	Add a star.
`Graph.add_path` (nlist[, data])	Add a path.
`Graph.add_cycle` (nlist[, data])	Add a cycle.
`Graph.clear` ()	Remove all nodes and edges.

networkx.Graph.add_node

add_node (*n*)

Add a single node n.

Parameters **n** : node

A node n can be any hashable Python object except None.

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.

Notes

On many platforms hashable items also include mutables such as Graphs, though one should be careful that the hash doesn't change on mutables.

Examples

```
>>> G=nx.Graph()
>>> G.add_node(1)
>>> G.add_node('Hello')
>>> K3=nx.complete_graph(3)
>>> G.add_node(K3)
>>> G.number_of_nodes()
3
```

networkx.Graph.add_nodes_from

add_nodes_from(*nbunch*)

Add nodes from nbunch.

Parameters **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.

Examples

```
>>> G=nx.Graph()
>>> G.add_nodes_from('Hello')
>>> K3=nx.complete_graph(3)
>>> G.add_nodes_from(K3)
>>> sorted(G.nodes())
[0, 1, 2, 'H', 'e', 'l', 'o']
```

networkx.Graph.remove_node

remove_node(*n*)

Remove node n.

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

Examples

```
>>> G=nx.complete_graph(3)  # complete graph on 3 nodes, K3
>>> G.edges()
[(0, 1), (0, 2), (1, 2)]
>>> G.remove_node(1)
>>> G.edges()
[(0, 2)]
```

networkx.Graph.remove_nodes_from

remove_nodes_from(*nbunch*)

Remove nodes specified in nbunch.

Parameters **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.

Examples

```
>>> G=nx.complete_graph(3)  # complete graph on 3 nodes, K3
>>> e=G.nodes()
>>> e
[0, 1, 2]
>>> G.remove_nodes_from(e)
>>> G.nodes()
[]
```

networkx.Graph.add_edge

add_edge(*u, v, data=1*)

Add an edge between u and v with optional data.

The nodes u and v will be automatically added if they are not already in the graph.

Parameters **u,v** : nodes

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

data : Python object

Edge data (or labels or objects) can be entered via the optional argument data which defaults to 1.

Some NetworkX algorithms are designed for weighted graphs for which the edge data must be a number. These may behave unpredictably for edge data that isn't a number.

See Also:

`Parallel`

Notes

Adding an edge that already exists *overwrites* the edgedata.

Examples

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

```
>>> G=nx.Graph()
>>> 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 the data myedge to the edge (1,2).

```
>>> myedge=1.3
>>> G.add_edge(1, 2, myedge)
```

networkx.Graph.add_edges_from

add_edges_from(*ebunch, data=1*)

Add all the edges in ebunch.

Parameters ebunch : list or container of edges

The container must be iterable or an iterator. It is iterated over once. Adding the same edge twice has no effect and does not raise an exception. The edges in ebunch must be 2-tuples (u,v) or 3-tuples (u,v,d).

data : any Python object The default data for edges with no data given. If unspecified the integer 1 will be used.

See Also:

add_edge

Examples

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

networkx.Graph.remove_edge

remove_edge(*u, v*)

Remove the edge between (u,v).

Parameters u,v: nodes :

See Also:

remove_edges_from remove a collection of edges

Examples

```
>>> G=nx.path_graph(4)
>>> G.remove_edge(0,1)
>>> e=(1,2)
>>> G.remove_edge(*e) # unpacks e from an edge tuple
>>> e=(2,3,'data')
>>> G.remove_edge(*e[:2]) # edge tuple with data
```

networkx.Graph.remove_edges_from

remove_edges_from(*ebunch*)

Remove all edges specified in ebunch.

Parameters ebunch: list or container of edge tuples :

A container of edge 2-tuples (u,v) or edge 3-tuples(u,v,d) though d is ignored unless we are a multigraph.

See Also:

remove_edge

Notes

Will fail silently if the edge (u,v) is not in the graph.

Examples

```
>>> G=nx.path_graph(4)
>>> ebunch=[(1,2),(2,3)]
>>> G.remove_edges_from(ebunch)
```

networkx.Graph.add_star

add_star (*nlist, data=None*)

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.

data : list or iterable, optional

Data to add to the edges in the path. The length should be one less than len(nlist).

Examples

```
>>> G=nx.Graph()
>>> G.add_star([0,1,2,3])
>>> G.add_star([10,11,12],data=['a','b'])
```

networkx.Graph.add_path

add_path (*nlist, data=None*)

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.

data : list or iterable, optional

Data to add to the edges in the path. The length should be one less than len(nlist).

Examples

```
>>> G=nx.Graph()
>>> G.add_path([0,1,2,3])
>>> G.add_path([10,11,12],data=['a','b'])
```

networkx.Graph.add_cycle

add_cycle(*nlist, data=None*)

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.

data : list or iterable, optional

Data to add to the edges in the path. The length should be the same as nlist.

Examples

```
>>> G=nx.Graph()
>>> G.add_cycle([0,1,2,3])
>>> G.add_cycle([10,11,12],data=['a','b','c'])
```

networkx.Graph.clear

clear()

Remove all nodes and edges.

This also removes the name.

Examples

```
>>> G=nx.path_graph(4)
>>> G.clear()
>>> G.nodes()
[]
>>> G.edges()
[]
```

Iterating over nodes and edges

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

networkx.Graph.nodes

nodes ()

Return a list of the nodes.

Examples

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

networkx.Graph.nodes_iter

nodes_iter ()

Return an iterator for the nodes.

Notes

It is simpler and equivalent to use the expression "for n in G"

```
>>> G=nx.path_graph(3)
>>> for n in G:
```

```
...     print n,
0 1 2
```

Examples

```
>>> G=nx.path_graph(3)
>>> for n in G.nodes_iter():
...     print n,
0 1 2
```

You can also say

```
>>> G=nx.path_graph(3)
>>> for n in G:
...     print n,
0 1 2
```

networkx.Graph.__iter__

__iter__()

Iterate over the nodes. Use "for n in G".

Examples

```
>>> G=nx.path_graph(4)
>>> print [n for n in G]
[0, 1, 2, 3]
```

networkx.Graph.edges

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

Return a list of edges.

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

data : bool

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

Returns Edges that are adjacent to any node in nbunch, :

or a list of all edges if nbunch is not specified. :

Examples

```
>>> G=nx.path_graph(4)
>>> G.edges()
[(0, 1), (1, 2), (2, 3)]
>>> G.edges(data=True) # default edge data is 1
[(0, 1, 1), (1, 2, 1), (2, 3, 1)]
```

networkx.Graph.edges_iter

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

Return an iterator over the edges.

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

data : bool

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

Returns An iterator over edges that are adjacent to any node in nbunch, :

or over all edges if nbunch is not specified. :

Examples

```
>>> G=nx.path_graph(4)
>>> G.edges()
[(0, 1), (1, 2), (2, 3)]
>>> G.edges(data=True) # default edge data is 1
[(0, 1, 1), (1, 2, 1), (2, 3, 1)]
```

networkx.Graph.get_edge

get_edge (*u, v, default=None*)

Return the data associated with the edge (u,v).

Parameters u,v : nodes

If u or v are not nodes in graph an exception is raised.

default: any Python object :

Value to return if the edge (u,v) is not found. If not specified, raise an exception.

Notes

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

```
>>> G[0][1]
1
```

Examples

```
>>> G=nx.path_graph(4) # path graph with edge data all set to 1
>>> G.get_edge(0,1)
1
>>> e=(0,1)
>>> G.get_edge(*e) # tuple form
1
```

networkx.Graph.neighbors

neighbors (*n*)

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

Notes

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

```
>>> G=nx.Graph()
>>> G.add_edge('a','b','data')
>>> G['a']
{'b': 'data'}
```

Examples

```
>>> G=nx.path_graph(4)
>>> 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 iterate over the using the idiom >>> print [n for n in G[0]] [1]

Examples

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

networkx.Graph.__getitem__

__getitem__(*n*)
Return the neighbors of node n. Use "G[n]".

Notes

G[n] is similar to G.neighbors(n) but the internal data dictionary is returned instead of a list.

G[u][v] returns the edge data for edge (u,v).

```
>>> G=nx.path_graph(4)
>>> print G[0][1]
1
```

Assigning G[u][v] may corrupt the graph data structure.

Examples

```
>>> G=nx.path_graph(4)
>>> print G[0]
{1: 1}
```

networkx.Graph.adjacency_list

adjacency_list()

Return an adjacency list as a Python list of lists

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

Examples

```
>>> G=nx.path_graph(4)
>>> G.adjacency_list() # in sorted node order 0,1,2,3
[[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.

Notes

The dictionary returned is part of the internal graph data structure; changing it could corrupt that structure. This is meant for fast inspection, not mutation.

For MultiGraph/MultiDiGraph multigraphs, a list of edge data is the value in the dict.

Examples

```
>>> G=nx.path_graph(4)
>>> [(n,nbrdict) for n,nbrdict in G.adjacency_iter()]
[(0, {1: 1}), (1, {0: 1, 2: 1}), (2, {1: 1, 3: 1}), (3, {2: 1})]
```

networkx.Graph.nbunch_iter

nbunch_iter(*nbunch=None*)

Return an iterator of nodes contained in nbunch that are also in the graph.

Parameters **nbunch** : list, iterable

A container of nodes that will be iterated through once (thus it should be an iterator or be iterable). Each element of the container should be a valid node type: any hashable type except None. If nbunch is None, return all edges data in the graph. Nodes in nbunch that are not in the graph will be (quietly) ignored.

Notes

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 values returned by an iterator nbunch is not hashable, a NetworkXError is raised.

Information about graph structure

`Graph.has_node` (n)	Return True if graph has node n.
`Graph.__contains__` (n)	Return True if n is a node, False otherwise. Use "n in G".
`Graph.has_edge` (u, v)	Return True if graph contains the edge (u,v), False otherwise.
`Graph.has_neighbor` (u, v)	Return True if node u has neighbor v.
`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.
`Graph.number_of_nodes` ()	Return the number of nodes.
`Graph.__len__` ()	Return the number of nodes. Use "len(G)".
`Graph.size` ([weighted])	Return the number of edges.
`Graph.number_of_edges` ([u, v])	Return the number of edges between two nodes.
`Graph.number_of_selfloops` ()	Return the number of selfloop edges (edge from a node to itself).
`Graph.degree` ([nbunch, with_labels, ...])	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 graph has node n.

Notes

It is more readable and simpler to use >>> 0 in G True

Examples

```
>>> G=nx.path_graph(4)
>>> print G.has_node(0)
True
```

networkx.Graph.__contains__

__contains__(*n*)

Return True if n is a node, False otherwise. Use "n in G".

Examples

```
>>> G=nx.path_graph(4)
>>> print 1 in G
True
```

networkx.Graph.has_edge

has_edge(*u, v*)

Return True if graph contains the edge (u,v), False otherwise.

See Also:

`Graph.has_neighbor`

Examples

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

```
>>> G=nx.path_graph(4)
>>> G.has_edge(0,1)  # called using two nodes
True
>>> e=(0,1)
>>> G.has_edge(*e)  #  e is a 2-tuple (u,v)
True
>>> e=(0,1,'data')
>>> G.has_edge(*e[:2])  # e is a 3-tuple (u,v,d)
True
```

The following syntax are all equivalent:

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

networkx.Graph.has_neighbor

has_neighbor(*u, v*)

Return True if node u has neighbor v.

This returns True if there exists any edge (u,v,data) for some data.

See Also:

has_edge

Examples

```
>>> G=nx.path_graph(4)
>>> G.has_neighbor(0,1)
True
>>> G.has_edge(0,1) # same as has_neighbor
True
>>> 1 in G[0]   # this gives KeyError if u not in G
True
```

networkx.Graph.nodes_with_selfloops

nodes_with_selfloops()

Return a list of nodes with self loops.

Examples

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

Parameters data : bool

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

Examples

```
>>> G=nx.Graph()
>>> G.add_edge(1,1)
>>> G.add_edge(1,2)
>>> G.selfloop_edges()
[(1, 1)]
>>> G.selfloop_edges(data=True)
[(1, 1, 1)]
```

networkx.Graph.order

order()

Return the number of nodes.

See Also:

Graph.order, Graph.__len__

networkx.Graph.number_of_nodes

number_of_nodes()

Return the number of nodes.

Notes

This is the same as

```
>>> len(G)
4
```

and

```
>>> G.order()
4
```

Examples

```
>>> G=nx.path_graph(4)
>>> print len(G)
4
```

networkx.Graph.__len__

__len__()

Return the number of nodes. Use "len(G)".

Examples

```
>>> G=nx.path_graph(4)
>>> print len(G)
4
```

networkx.Graph.size

size(*weighted=False*)

Return the number of edges.

Parameters weighted : bool, optional

If True return the sum of the edge weights.

See Also:

Graph.number_of_edges

Examples

```
>>> G=nx.path_graph(4)
>>> G.size()
3
```

```
>>> G=nx.Graph()
>>> G.add_edge('a','b',2)
>>> G.add_edge('b','c',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

If u and v are specified, return the number of edges between u and v. Otherwise return the total number of all edges.

Examples

```
>>> G=nx.path_graph(4)
>>> 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 (edge from a node to itself).

Examples

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

with_labels : False | True
Return a list of degrees (False) or a dictionary of degrees keyed by node (True).

weighted : False | True
If the graph is weighted return the weighted degree (the sum of edge weights).

Examples

```
>>> G=nx.path_graph(4)
>>> 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 that node.

Parameters 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.

weighted : False | True
If the graph is weighted return the weighted degree (the sum of edge weights).

Examples

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

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

Notes

This makes a complete of the graph but does not make copies of any underlying node or edge data. The node and edge data in the copy still point to the same objects as in the original.

networkx.Graph.to_directed

`to_directed``()`

Return a directed representation of the graph.

A new directed graph is returned 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).

networkx.Graph.subgraph

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

Return the subgraph induced on nodes in nbunch.

Parameters **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.

DiGraph

Overview

class **`DiGraph`**`(`*data=None, name='', weighted=True*`)`

A directed graph that allows self-loops, but not multiple (parallel) edges.

Edge and node data is the same as for Graph. Subclass of Graph.

An empty digraph is created with

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

Adding and Removing Nodes and Edges

`DiGraph.add_node` (n)	Add a single node n.
`DiGraph.add_nodes_from` (nbunch)	Add nodes from nbunch.
`DiGraph.remove_node` (n)	Remove node n.
`DiGraph.remove_nodes_from` (nbunch)	Remove nodes specified in nbunch.
`DiGraph.add_edge` (u, v[, data])	Add an edge between u and v with optional data.
`DiGraph.add_edges_from` (ebunch[, data])	Add all the edges in ebunch.
`DiGraph.remove_edge` (u, v)	Remove the edge between (u,v).
`DiGraph.remove_edges_from` (ebunch)	Remove all edges specified in ebunch.
`DiGraph.add_star` (nlist[, data])	Add a star.
`DiGraph.add_path` (nlist[, data])	Add a path.
`DiGraph.add_cycle` (nlist[, data])	Add a cycle.
`DiGraph.clear` ()	Remove all nodes and edges.

networkx.DiGraph.add_node

add_node (*n*)

Add a single node n.

Parameters **n** : node

A node n can be any hashable Python object except None.

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.

Notes

On many platforms hashable items also include mutables such as DiGraphs, though one should be careful that the hash doesn't change on mutables.

Examples

```
>>> G=nx.DiGraph()
>>> G.add_node(1)
>>> G.add_node('Hello')
>>> K3=nx.complete_graph(3)
>>> G.add_node(K3)
>>> G.number_of_nodes()
3
```

networkx.DiGraph.add_nodes_from

add_nodes_from(*nbunch*)

Add nodes from nbunch.

Parameters 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.

Examples

```
>>> G=nx.DiGraph()
>>> G.add_nodes_from('Hello')
>>> K3=nx.complete_graph(3)
>>> G.add_nodes_from(K3)
>>> sorted(G.nodes())
[0, 1, 2, 'H', 'e', 'l', 'o']
```

networkx.DiGraph.remove_node

remove_node(*n*)

Remove node n.

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

Examples

```
>>> G=nx.complete_graph(3)  # complete graph on 3 nodes, K3
>>> G.edges()
[(0, 1), (0, 2), (1, 2)]
>>> G.remove_node(1)
>>> G.edges()
[(0, 2)]
```

networkx.DiGraph.remove_nodes_from

remove_nodes_from(*nbunch*)

Remove nodes specified in nbunch.

Parameters 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.

Examples

```
>>> G=nx.complete_graph(3)  # complete graph on 3 nodes, K3
>>> e=G.nodes()
>>> e
[0, 1, 2]
>>> G.remove_nodes_from(e)
```

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

networkx.DiGraph.add_edge

add_edge (*u, v, data=1*)

Add an edge between u and v with optional data.

The nodes u and v will be automatically added if they are not already in the graph.

Parameters u,v : nodes

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

data : Python object

Edge data (or labels or objects) can be entered via the optional argument data which defaults to 1.

Some NetworkX algorithms are designed for weighted graphs for which the edge data must be a number. These may behave unpredictably for edge data that isn't a number.

See Also:

`Parallel`

Notes

Adding an edge that already exists *overwrites* the edgedata.

Examples

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

```
>>> G=nx.DiGraph()
>>> 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 the data myedge to the edge (1,2).

```
>>> myedge=1.3
>>> G.add_edge(1, 2, myedge)
```

networkx.DiGraph.add_edges_from

add_edges_from (*ebunch, data=1*)

Add all the edges in ebunch.

Parameters ebunch : list or container of edges

The container must be iterable or an iterator. It is iterated over once. Adding the same edge twice has no effect and does not raise an exception. The edges in ebunch must be 2-tuples (u,v) or 3-tuples (u,v,d).

data : any Python object The default data for edges with no data given. If unspecified the integer 1 will be used.

See Also:

add_edge

Examples

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

networkx.DiGraph.remove_edge

remove_edge (*u, v*)

Remove the edge between (u,v).

Parameters u,v: nodes :

See Also:

remove_edges_from remove a collection of edges

Examples

```
>>> G=nx.path_graph(4)
>>> G.remove_edge(0,1)
>>> e=(1,2)
>>> G.remove_edge(*e) # unpacks e from an edge tuple
>>> e=(2,3,'data')
>>> G.remove_edge(*e[:2]) # edge tuple with data
```

networkx.DiGraph.remove_edges_from

remove_edges_from (*ebunch*)

Remove all edges specified in ebunch.

Parameters ebunch: list or container of edge tuples :

A container of edge 2-tuples (u,v) or edge 3-tuples(u,v,d) though d is ignored unless we are a multigraph.

See Also:

remove_edge

Notes

Will fail silently if the edge (u,v) is not in the graph.

Examples

```
>>> G=nx.path_graph(4)
>>> ebunch=[(1,2),(2,3)]
>>> G.remove_edges_from(ebunch)
```

networkx.DiGraph.add_star

add_star (*nlist, data=None*)

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.

data : list or iterable, optional

Data to add to the edges in the path. The length should be one less than len(nlist).

Examples

```
>>> G=nx.Graph()
>>> G.add_star([0,1,2,3])
>>> G.add_star([10,11,12],data=['a','b'])
```

networkx.DiGraph.add_path

add_path (*nlist, data=None*)

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.

data : list or iterable, optional

Data to add to the edges in the path. The length should be one less than len(nlist).

Examples

```
>>> G=nx.Graph()
>>> G.add_path([0,1,2,3])
>>> G.add_path([10,11,12],data=['a','b'])
```

networkx.DiGraph.add_cycle

add_cycle (*nlist, data=None*)

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.

data : list or iterable, optional

Data to add to the edges in the path. The length should be the same as nlist.

Examples

```
>>> G=nx.Graph()
>>> G.add_cycle([0,1,2,3])
>>> G.add_cycle([10,11,12],data=['a','b','c'])
```

networkx.DiGraph.clear

clear()

Remove all nodes and edges.

This also removes the name.

Examples

```
>>> G=nx.path_graph(4)
>>> G.clear()
>>> G.nodes()
[]
>>> G.edges()
[]
```

Iterating over nodes and edges

`DiGraph.nodes` ()	Return a list of the nodes.
`DiGraph.nodes_iter` ()	Return an iterator for the nodes.
`DiGraph.__iter__` ()	Iterate over the nodes. Use "for n in G".
`DiGraph.edges` ([nbunch, data])	Return a list of edges.
`DiGraph.edges_iter` ([nbunch, data])	Return an iterator over the edges.
`DiGraph.get_edge` (u, v[, default])	Return the data associated with the edge (u,v).
`DiGraph.neighbors` (n)	Return a list of the nodes connected to the node n.
`DiGraph.neighbors_iter` (n)	Return an iterator over all neighbors of node n.
`DiGraph.__getitem__` (n)	Return the neighbors of node n. Use "G[n]".
`DiGraph.successors` (n)	Return a list of the nodes connected to the node n.
`DiGraph.successors_iter` (n)	Return an iterator over all neighbors of node 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 as a Python list of lists
`DiGraph.adjacency_iter` ()	Return an iterator of (node, adjacency dict) tuples for all nodes.
`DiGraph.nbunch_iter` ([nbunch])	Return an iterator of nodes contained in nbunch that are also in the graph.

networkx.DiGraph.nodes

nodes ()

Return a list of the nodes.

Examples

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

networkx.DiGraph.nodes_iter

nodes_iter()
Return an iterator for the nodes.

Notes

It is simpler and equivalent to use the expression "for n in G"

```
>>> G=nx.path_graph(3)
>>> for n in G:
...     print n,
0 1 2
```

Examples

```
>>> G=nx.path_graph(3)
>>> for n in G.nodes_iter():
...     print n,
0 1 2
```

You can also say

```
>>> G=nx.path_graph(3)
>>> for n in G:
...     print n,
0 1 2
```

networkx.DiGraph.__iter__

__iter__()
Iterate over the nodes. Use "for n in G".

Examples

```
>>> G=nx.path_graph(4)
>>> print [n for n in G]
[0, 1, 2, 3]
```

networkx.DiGraph.edges

edges(*nbunch=None, data=False*)
Return a list of edges.

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

data : bool

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

Returns **Edges that are adjacent to any node in nbunch,** :

or a list of all edges if nbunch is not specified. :

Examples

```
>>> G=nx.path_graph(4)
>>> G.edges()
[(0, 1), (1, 2), (2, 3)]
>>> G.edges(data=True) # default edge data is 1
[(0, 1, 1), (1, 2, 1), (2, 3, 1)]
```

networkx.DiGraph.edges_iter

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

Return an iterator over the edges.

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

data : bool

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

Returns **An iterator over edges that are adjacent to any node in nbunch,** :

or over all edges if nbunch is not specified. :

Examples

```
>>> G=nx.path_graph(4)
>>> G.edges()
[(0, 1), (1, 2), (2, 3)]
>>> G.edges(data=True) # default edge data is 1
[(0, 1, 1), (1, 2, 1), (2, 3, 1)]
```

networkx.DiGraph.get_edge

get_edge (*u, v, default=None*)

Return the data associated with the edge (u,v).

Parameters **u,v** : nodes

If u or v are not nodes in graph an exception is raised.

default: any Python object :

Value to return if the edge (u,v) is not found. If not specified, raise an exception.

Notes

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

```
>>> G[0][1]
1
```

Examples

```
>>> G=nx.path_graph(4) # path graph with edge data all set to 1
>>> G.get_edge(0,1)
1
>>> e=(0,1)
>>> G.get_edge(*e) # tuple form
1
```

networkx.DiGraph.neighbors

neighbors(*n*)

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

Notes

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

```
>>> G=nx.Graph()
>>> G.add_edge('a','b','data')
>>> G['a']
{'b': 'data'}
```

Examples

```
>>> G=nx.path_graph(4)
>>> G.neighbors(0)
[1]
```

networkx.DiGraph.neighbors_iter

neighbors_iter(*n*)

Return an iterator over all neighbors of node n.

Notes

It is faster to iterate over the using the idiom >>> print [n for n in G[0]] [1]

Examples

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

networkx.DiGraph.__getitem__

__getitem__(*n*)

Return the neighbors of node n. Use “G[n]”.

Notes

G[n] is similar to G.neighbors(n) but the internal data dictionary is returned instead of a list.

G[u][v] returns the edge data for edge (u,v).

```
>>> G=nx.path_graph(4)
>>> print G[0][1]
1
```

Assigning G[u][v] may corrupt the graph data structure.

Examples

```
>>> G=nx.path_graph(4)
>>> print G[0]
{1: 1}
```

networkx.DiGraph.successors

successors (*n*)

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

Notes

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

```
>>> G=nx.Graph()
>>> G.add_edge('a','b','data')
>>> G['a']
{'b': 'data'}
```

Examples

```
>>> G=nx.path_graph(4)
>>> G.neighbors(0)
[1]
```

networkx.DiGraph.successors_iter

successors_iter (*n*)

Return an iterator over all neighbors of node n.

Notes

It is faster to iterate over the using the idiom >>> print [n for n in G[0]] [1]

Examples

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

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 as a Python list of lists

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

Examples

```
>>> G=nx.path_graph(4)
>>> G.adjacency_list() # in sorted node order 0,1,2,3
[[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.

Notes

The dictionary returned is part of the internal graph data structure; changing it could corrupt that structure. This is meant for fast inspection, not mutation.

For MultiGraph/MultiDiGraph multigraphs, a list of edge data is the value in the dict.

Examples

```
>>> G=nx.path_graph(4)
>>> [(n,nbrdict) for n,nbrdict in G.adjacency_iter()]
[(0, {1: 1}), (1, {0: 1, 2: 1}), (2, {1: 1, 3: 1}), (3, {2: 1})]
```

networkx.DiGraph.nbunch_iter

`nbunch_iter`(*nbunch=None*)

Return an iterator of nodes contained in nbunch that are also in the graph.

Parameters **nbunch** : list, iterable

A container of nodes that will be iterated through once (thus it should be an iterator or be iterable). Each element of the container should be a valid node type: any hashable type except None. If nbunch is None, return all edges data in the graph. Nodes in nbunch that are not in the graph will be (quietly) ignored.

Notes

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 values returned by an iterator nbunch is not hashable, a NetworkXError is raised.

Information about graph structure

DiGraph.has_node (n)	Return True if graph has node n.
DiGraph.__contains__ (n)	Return True if n is a node, False otherwise. Use "n in G".
DiGraph.has_edge (u, v)	Return True if graph contains the edge (u,v), False otherwise.
DiGraph.has_neighbor (u, v)	Return True if node u has neighbor v.
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.
DiGraph.number_of_nodes ()	Return the number of nodes.
DiGraph.__len__ ()	Return the number of nodes. Use "len(G)".
DiGraph.size ([weighted])	Return the number of edges.
DiGraph.number_of_edges ([u, v])	Return the number of edges between two nodes.
DiGraph.number_of_selfloops ()	Return the number of selfloop edges (edge from a node to itself).
DiGraph.degree ([nbunch, with_labels, ...])	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 graph has node n.

Notes

It is more readable and simpler to use >>> 0 in G True

Examples

```
>>> G=nx.path_graph(4)
>>> print G.has_node(0)
True
```

networkx.DiGraph.__contains__

__contains__ (*n*)

Return True if n is a node, False otherwise. Use "n in G".

Examples

```
>>> G=nx.path_graph(4)
>>> print 1 in G
True
```

networkx.DiGraph.has_edge

has_edge (*u*, *v*)

Return True if graph contains the edge (u,v), False otherwise.

See Also:

Graph.has_neighbor

Examples

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

```
>>> G=nx.path_graph(4)
>>> G.has_edge(0,1)  # called using two nodes
True
>>> e=(0,1)
>>> G.has_edge(*e)  #  e is a 2-tuple (u,v)
True
>>> e=(0,1,'data')
>>> G.has_edge(*e[:2])  # e is a 3-tuple (u,v,d)
True
```

The following syntax are all equivalent:

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

networkx.DiGraph.has_neighbor

has_neighbor (*u*, *v*)

Return True if node u has neighbor v.

This returns True if there exists any edge (u,v,data) for some data.

See Also:

has_edge

Examples

```
>>> G=nx.path_graph(4)
>>> G.has_neighbor(0,1)
True
>>> G.has_edge(0,1) # same as has_neighbor
True
>>> 1 in G[0]  # this gives KeyError if u not in G
True
```

networkx.DiGraph.nodes_with_selfloops
nodes_with_selfloops()

Return a list of nodes with self loops.

Examples

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

Parameters data : bool

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

Examples

```
>>> G=nx.Graph()
>>> G.add_edge(1,1)
>>> G.add_edge(1,2)
>>> G.selfloop_edges()
[(1, 1)]
>>> G.selfloop_edges(data=True)
[(1, 1, 1)]
```

networkx.DiGraph.order
order()

Return the number of nodes.

See Also:

Graph.order, Graph.__len__

networkx.DiGraph.number_of_nodes
number_of_nodes()

Return the number of nodes.

Notes

This is the same as

```
>>> len(G)
4
```

and

```
>>> G.order()
4
```

Examples

```
>>> G=nx.path_graph(4)
>>> print len(G)
4
```

networkx.DiGraph.__len__

__len__()

Return the number of nodes. Use "len(G)".

Examples

```
>>> G=nx.path_graph(4)
>>> print len(G)
4
```

networkx.DiGraph.size

size(*weighted=False*)

Return the number of edges.

Parameters **weighted** : bool, optional

If True return the sum of the edge weights.

See Also:

Graph.number_of_edges

Examples

```
>>> G=nx.path_graph(4)
>>> G.size()
3
```

```
>>> G=nx.Graph()
>>> G.add_edge('a','b',2)
>>> G.add_edge('b','c',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

If u and v are specified, return the number of edges between u and v. Otherwise return the total number of all edges.

Examples

```
>>> G=nx.path_graph(4)
>>> 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 (edge from a node to itself).

Examples

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

with_labels : False | True

Return a list of degrees (False) or a dictionary of degrees keyed by node (True).

weighted : False | True

If the graph is weighted return the weighted degree (the sum of edge weights).

Examples

```
>>> G=nx.path_graph(4)
>>> 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 that node.

Parameters **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.

weighted : False | True

If the graph is weighted return the weighted degree (the sum of edge weights).

Examples

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

Notes

This makes a complete of the graph but does not make copies of any underlying node or edge data. The node and edge data in the copy still point to the same objects as in the original.

networkx.DiGraph.to_undirected

`to_undirected`()

Return an undirected representation of the digraph.

A new graph is returned 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.

networkx.DiGraph.subgraph

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

Return the subgraph induced on nodes in nbunch.

Parameters **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.

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.

3.2.2 Multigraphs

The MultiGraph and MultiDiGraph classes extend the basic graphs by allowing multiple (parallel) edges between nodes.

MultiGraph

Overview

class **`MultiGraph`**(*data=None, name='', weighted=True*)

An undirected graph that allows multiple (parallel) edges with arbitrary data on the edges.

Subclass of Graph.

An empty multigraph is created with

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

Examples

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

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

You can add nodes in the same way as the simple Graph class >>> G.add_nodes_from(xrange(100,110))

You can add edges with data/labels/objects as for the Graph class, but here the same two nodes can have more than one edge between them.

```
>>> G.add_edges_from([(1,2,0.776),(1,2,0.535)])
```

See also the MultiDiGraph class for a directed graph version.

MultiGraph inherits from Graph, overriding the following methods:

- add_edge
- add_edges_from
- remove_edge
- remove_edges_from
- edges_iter
- get_edge
- degree_iter
- selfloop_edges
- number_of_selfloops
- number_of_edges
- to_directed
- subgraph

Adding and Removing Nodes and Edges

MultiGraph.add_node (n)	Add a single node n.
MultiGraph.add_nodes_from (nbunch)	Add nodes from nbunch.
MultiGraph.remove_node (n)	Remove node n.
MultiGraph.remove_nodes_from (nbunch)	Remove nodes specified in nbunch.
MultiGraph.add_edge (u, v[, data])	Add an edge between u and v with optional data.
MultiGraph.add_edges_from (ebunch[, data])	Add all the edges in ebunch.
MultiGraph.remove_edge (u, v[, data])	Remove the edge between (u,v).
MultiGraph.remove_edges_from (ebunch)	Remove all edges specified in ebunch.
MultiGraph.add_star (nlist[, data])	Add a star.
MultiGraph.add_path (nlist[, data])	Add a path.
MultiGraph.add_cycle (nlist[, data])	Add a cycle.
MultiGraph.clear ()	Remove all nodes and edges.

networkx.MultiGraph.add_node

add_node (*n*)

Add a single node n.

Parameters **n** : node

A node n can be any hashable Python object except None.

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.

Notes

On many platforms hashable items also include mutables such as Graphs, though one should be careful that the hash doesn't change on mutables.

Examples

```
>>> G=nx.Graph()
>>> G.add_node(1)
>>> G.add_node('Hello')
>>> K3=nx.complete_graph(3)
>>> G.add_node(K3)
>>> G.number_of_nodes()
3
```

networkx.MultiGraph.add_nodes_from

add_nodes_from(*nbunch*)

Add nodes from nbunch.

Parameters 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.

Examples

```
>>> G=nx.Graph()
>>> G.add_nodes_from('Hello')
>>> K3=nx.complete_graph(3)
>>> G.add_nodes_from(K3)
>>> sorted(G.nodes())
[0, 1, 2, 'H', 'e', 'l', 'o']
```

networkx.MultiGraph.remove_node

remove_node(*n*)

Remove node n.

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

Examples

```
>>> G=nx.complete_graph(3)  # complete graph on 3 nodes, K3
>>> G.edges()
[(0, 1), (0, 2), (1, 2)]
>>> G.remove_node(1)
>>> G.edges()
[(0, 2)]
```

networkx.MultiGraph.remove_nodes_from

remove_nodes_from(*nbunch*)

Remove nodes specified in nbunch.

Parameters 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.

Examples

```
>>> G=nx.complete_graph(3)  # complete graph on 3 nodes, K3
>>> e=G.nodes()
>>> e
[0, 1, 2]
>>> G.remove_nodes_from(e)
```

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

networkx.MultiGraph.add_edge

add_edge(*u, v, data=1*)

Add an edge between u and v with optional data.

The nodes u and v will be automatically added if they are not already in the graph.

Parameters u,v : nodes

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

data : Python object

Edge data (or labels or objects) can be entered via the optional argument data which defaults to 1.

Some NetworkX algorithms are designed for weighted graphs for which the edge data must be a number. These may behave unpredictably for edge data that isn't a number.

See Also:

`Parallel`

Notes

Adding an edge that already exists *overwrites* the edgedata.

Examples

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

```
>>> G=nx.Graph()
>>> 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 the data myedge to the edge (1,2).

```
>>> myedge=1.3
>>> G.add_edge(1, 2, myedge)
```

networkx.MultiGraph.add_edges_from

add_edges_from(*ebunch, data=1*)

Add all the edges in ebunch.

Parameters ebunch : list or container of edges

The container must be iterable or an iterator. It is iterated over once. Adding the same edge twice has no effect and does not raise an exception. The edges in ebunch must be 2-tuples (u,v) or 3-tuples (u,v,d).

data : any Python object The default data for edges with no data given. If unspecified the integer 1 will be used.

See Also:

`add_edge`

Examples

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

networkx.MultiGraph.remove_edge

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

Remove the edge between (u,v).

If d is defined only remove the first edge found with edgedata == d.

If d is None, remove all edges between u and v.

networkx.MultiGraph.remove_edges_from

remove_edges_from (*ebunch*)

Remove all edges specified in ebunch.

Parameters ebunch: list or container of edge tuples :

A container of edge 2-tuples (u,v) or edge 3-tuples(u,v,d) though d is ignored unless we are a multigraph.

See Also:

`remove_edge`

Notes

Will fail silently if the edge (u,v) is not in the graph.

Examples

```
>>> G=nx.path_graph(4)
>>> ebunch=[(1,2),(2,3)]
>>> G.remove_edges_from(ebunch)
```

networkx.MultiGraph.add_star

add_star (*nlist, data=None*)

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.

data : list or iterable, optional

Data to add to the edges in the path. The length should be one less than len(nlist).

Examples

```
>>> G=nx.Graph()
>>> G.add_star([0,1,2,3])
>>> G.add_star([10,11,12],data=['a','b'])
```

networkx.MultiGraph.add_path

add_path (*nlist, data=None*)

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.

data : list or iterable, optional

Data to add to the edges in the path. The length should be one less than len(nlist).

Examples

```
>>> G=nx.Graph()
>>> G.add_path([0,1,2,3])
>>> G.add_path([10,11,12],data=['a','b'])
```

networkx.MultiGraph.add_cycle

add_cycle (*nlist, data=None*)

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.

data : list or iterable, optional

Data to add to the edges in the path. The length should be the same as nlist.

Examples

```
>>> G=nx.Graph()
>>> G.add_cycle([0,1,2,3])
>>> G.add_cycle([10,11,12],data=['a','b','c'])
```

networkx.MultiGraph.clear

clear ()

Remove all nodes and edges.

This also removes the name.

Examples

```
>>> G=nx.path_graph(4)
>>> G.clear()
>>> G.nodes()
[]
>>> G.edges()
[]
```

Iterating over nodes and edges

MultiGraph.nodes ()	Return a list of the nodes.
MultiGraph.nodes_iter ()	Return an iterator for the nodes.
MultiGraph.__iter__ ()	Iterate over the nodes. Use "for n in G".
MultiGraph.edges ([nbunch, data])	Return a list of edges.
MultiGraph.edges_iter ([nbunch, data])	Return an iterator over the edges.
MultiGraph.get_edge (u, v[, no_edge])	Return a list of edge data for all edges between u and v.
MultiGraph.neighbors (n)	Return a list of the nodes connected to the node n.
MultiGraph.neighbors_iter (n)	Return an iterator over all neighbors of node n.
MultiGraph.__getitem__ (n)	Return the neighbors of node n. Use "G[n]".
MultiGraph.adjacency_list ()	Return an adjacency list as a Python list of lists
MultiGraph.adjacency_iter ()	Return an iterator of (node, adjacency dict) tuples for all nodes.
MultiGraph.nbunch_iter ([nbunch])	Return an iterator of nodes contained in nbunch that are also in the graph.

networkx.MultiGraph.nodes

nodes ()

Return a list of the nodes.

Examples

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

networkx.MultiGraph.nodes_iter

nodes_iter()

Return an iterator for the nodes.

Notes

It is simpler and equivalent to use the expression "for n in G"

```
>>> G=nx.path_graph(3)
>>> for n in G:
...     print n,
0 1 2
```

Examples

```
>>> G=nx.path_graph(3)
>>> for n in G.nodes_iter():
...     print n,
0 1 2
```

You can also say

```
>>> G=nx.path_graph(3)
>>> for n in G:
...     print n,
0 1 2
```

networkx.MultiGraph.__iter__

__iter__()

Iterate over the nodes. Use "for n in G".

Examples

```
>>> G=nx.path_graph(4)
>>> print [n for n in G]
[0, 1, 2, 3]
```

networkx.MultiGraph.edges

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

Return a list of edges.

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

data : bool

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

Returns **Edges that are adjacent to any node in nbunch,** :

or a list of all edges if nbunch is not specified. :

Examples

```
>>> G=nx.path_graph(4)
>>> G.edges()
[(0, 1), (1, 2), (2, 3)]
>>> G.edges(data=True) # default edge data is 1
[(0, 1, 1), (1, 2, 1), (2, 3, 1)]
```

networkx.MultiGraph.edges_iter

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

Return an iterator over the edges.

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

data : bool

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

Returns An iterator over edges that are adjacent to any node in nbunch, :

or over all edges if nbunch is not specified. :

Examples

```
>>> G=nx.path_graph(4)
>>> G.edges()
[(0, 1), (1, 2), (2, 3)]
>>> G.edges(data=True) # default edge data is 1
[(0, 1, 1), (1, 2, 1), (2, 3, 1)]
```

networkx.MultiGraph.get_edge

get_edge (*u, v, no_edge=None*)

Return a list of edge data for all edges between u and v.

If no_edge is specified and the edge (u,v) isn't found, (and u and v are nodes), return the value of no_edge. If no_edge is None (or u or v aren't nodes) raise an exception.

networkx.MultiGraph.neighbors

neighbors (*n*)

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

Notes

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

```
>>> G=nx.Graph()
>>> G.add_edge('a','b','data')
>>> G['a']
{'b': 'data'}
```

Examples

```
>>> G=nx.path_graph(4)
>>> 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 iterate over the using the idiom >>> print [n for n in G[0]] [1]

Examples

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

networkx.MultiGraph.__getitem__

__getitem__(*n*)

Return the neighbors of node n. Use "G[n]".

Notes

G[n] is similar to G.neighbors(n) but the internal data dictionary is returned instead of a list.

G[u][v] returns the edge data for edge (u,v).

```
>>> G=nx.path_graph(4)
>>> print G[0][1]
1
```

Assigning G[u][v] may corrupt the graph data structure.

Examples

```
>>> G=nx.path_graph(4)
>>> print G[0]
{1: 1}
```

networkx.MultiGraph.adjacency_list

adjacency_list()

Return an adjacency list as a Python list of lists

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

Examples

```
>>> G=nx.path_graph(4)
>>> G.adjacency_list() # in sorted node order 0,1,2,3
[[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.

Notes

The dictionary returned is part of the internal graph data structure; changing it could corrupt that structure. This is meant for fast inspection, not mutation.

For MultiGraph/MultiDiGraph multigraphs, a list of edge data is the value in the dict.

Examples

```
>>> G=nx.path_graph(4)
>>> [(n,nbrdict) for n,nbrdict in G.adjacency_iter()]
[(0, {1: 1}), (1, {0: 1, 2: 1}), (2, {1: 1, 3: 1}), (3, {2: 1})]
```

networkx.MultiGraph.nbunch_iter

nbunch_iter(*nbunch=None*)

Return an iterator of nodes contained in nbunch that are also in the graph.

Parameters **nbunch** : list, iterable

A container of nodes that will be iterated through once (thus it should be an iterator or be iterable). Each element of the container should be a valid node type: any hashable type except None. If nbunch is None, return all edges data in the graph. Nodes in nbunch that are not in the graph will be (quietly) ignored.

Notes

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 values returned by an iterator nbunch is not hashable, a NetworkXError is raised.

Information about graph structure

`MultiGraph.has_node` (n)	Return True if graph has node n.
`MultiGraph.__contains__` (n)	Return True if n is a node, False otherwise. Use "n in G".
`MultiGraph.has_edge` (u, v)	Return True if graph contains the edge (u,v), False otherwise.
`MultiGraph.has_neighbor` (u, v)	Return True if node u has neighbor v.
`MultiGraph.nodes_with_selfloops` ()	Return a list of nodes with self loops.
`MultiGraph.selfloop_edges` ()	Return a list of selfloop edges with data (3-tuples).
`MultiGraph.order` ()	Return the number of nodes.
`MultiGraph.number_of_nodes` ()	Return the number of nodes.
`MultiGraph.__len__` ()	Return the number of nodes. Use "len(G)".
`MultiGraph.size` ([weighted])	Return the number of edges.
`MultiGraph.number_of_edges` ([u, v])	Return the number of edges between two nodes.
`MultiGraph.number_of_selfloops` ()	Return the number of selfloop edges counting multiple 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 graph has node n.

Notes

It is more readable and simpler to use >>> 0 in G True

Examples

```
>>> G=nx.path_graph(4)
>>> print G.has_node(0)
True
```

networkx.MultiGraph.__contains__

__contains__(*n*)

Return True if n is a node, False otherwise. Use "n in G".

Examples

```
>>> G=nx.path_graph(4)
>>> print 1 in G
True
```

networkx.MultiGraph.has_edge

has_edge(*u, v*)

Return True if graph contains the edge (u,v), False otherwise.

See Also:

Graph.has_neighbor

Examples

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

```
>>> G=nx.path_graph(4)
>>> G.has_edge(0,1)   # called using two nodes
True
>>> e=(0,1)
>>> G.has_edge(*e)   #  e is a 2-tuple (u,v)
True
>>> e=(0,1,'data')
>>> G.has_edge(*e[:2])   # e is a 3-tuple (u,v,d)
True
```

The following syntax are all equivalent:

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

networkx.MultiGraph.has_neighbor

has_neighbor(*u, v*)

Return True if node u has neighbor v.

This returns True if there exists any edge (u,v,data) for some data.

See Also:

has_edge

Examples

```
>>> G=nx.path_graph(4)
>>> G.has_neighbor(0,1)
True
>>> G.has_edge(0,1) # same as has_neighbor
True
>>> 1 in G[0]  # this gives KeyError if u not in G
True
```

networkx.MultiGraph.nodes_with_selfloops

nodes_with_selfloops()

Return a list of nodes with self loops.

Examples

```
>>> G=nx.Graph()
>>> G.add_edge(1,1)
>>> G.add_edge(1,2)
>>> G.nodes_with_selfloops()
[1]
```

networkx.MultiGraph.selfloop_edges

selfloop_edges()

Return a list of selfloop edges with data (3-tuples).

networkx.MultiGraph.order

order()

Return the number of nodes.

See Also:

Graph.order, Graph.__len__

networkx.MultiGraph.number_of_nodes

number_of_nodes()

Return the number of nodes.

Notes

This is the same as

```
>>> len(G)
4
```

and

```
>>> G.order()
4
```

Examples

```
>>> G=nx.path_graph(4)
>>> print len(G)
4
```

networkx.MultiGraph.__len__

__len__()

Return the number of nodes. Use "len(G)".

Examples

```
>>> G=nx.path_graph(4)
>>> print len(G)
4
```

networkx.MultiGraph.size

size(*weighted=False*)

Return the number of edges.

Parameters weighted : bool, optional

If True return the sum of the edge weights.

See Also:

Graph.number_of_edges

Examples

```
>>> G=nx.path_graph(4)
>>> G.size()
3
```

```
>>> G=nx.Graph()
>>> G.add_edge('a','b',2)
>>> G.add_edge('b','c',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

If u and v are specified, return the number of edges between u and v. Otherwise return the total number of all edges.

Examples

```
>>> G=nx.path_graph(4)
>>> 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 counting multiple edges.

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** : 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.

with_labels : False | True

Return a list of degrees (False) or a dictionary of degrees keyed by node (True).

weighted : False | True

If the graph is weighted return the weighted degree (the sum of edge weights).

Examples

```
>>> G=nx.path_graph(4)
>>> 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 that node.

Parameters **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.

weighted : False | True
If the graph is weighted return the weighted degree (the sum of edge weights).

Examples

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

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

networkx.MultiGraph.copy

copy()
Return a copy of the graph.

Notes

This makes a complete of the graph but does not make copies of any underlying node or edge data. The node and edge data in the copy still point to the same objects as in the original.

networkx.MultiGraph.to_directed

to_directed()
Return a directed representation of the graph.

A new multidigraph is returned 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).

networkx.MultiGraph.subgraph

subgraph(*nbunch*, *copy=True*)
Return the subgraph induced on nodes in nbunch.

Parameters **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.

MultiDiGraph

Overview

class MultiDiGraph(*data=None, name='', weighted=True*)

A directed graph that allows multiple (parallel) edges with arbitrary data on the edges.

Subclass of DiGraph which is a subclass of Graph.

An empty multidigraph is created with

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

Examples

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

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

You can add nodes in the same way as the simple Graph class >>> G.add_nodes_from(xrange(100,110))

You can add edges with data/labels/objects as for the Graph class, but here the same two nodes can have more than one edge between them.

```
>>> G.add_edges_from([(1,2,0.776),(1,3,0.535)])
```

For graph coloring problems, one could use >>> G.add_edges_from([(1,2,"blue"),(1,3,"red")])

A MultiDiGraph edge is uniquely specified by a 3-tuple e=(u,v,x), where u and v are (hashable) objects (nodes) and x is an arbitrary (and not necessarily unique) object associated with that edge.

The graph is directed and multiple edges between the same nodes are allowed.

MultiDiGraph inherits from DiGraph, with all purely node-specific methods identical to those of DiGraph. MultiDiGraph edges are identical to MultiGraph edges, except that they are directed rather than undirected.

MultiDiGraph replaces the following DiGraph methods:

- add_edge
- add_edges_from
- remove_edge
- remove_edges_from
- get_edge
- edges_iter
- degree_iter
- in_degree_iter
- out_degree_iter
- selfloop_edges
- number_of_selfloops
- subgraph
- to_undirected

While MultiDiGraph does not inherit from MultiGraph, we compare them here. MultiDigraph adds the following methods to those of MultiGraph:

- has_successor
- has_predecessor
- successors
- predecessors
- successors_iter
- predecessors_iter
- in_degree
- out_degree
- in_degree_iter
- out_degree_iter
- reverse

Adding and Removing Nodes and Edges

`MultiDiGraph.add_node` (n)	Add a single node n.
`MultiDiGraph.add_nodes_from` (nbunch)	Add nodes from nbunch.
`MultiDiGraph.remove_node` (n)	Remove node n.
`MultiDiGraph.remove_nodes_from` (nbunch)	Remove nodes specified in nbunch.
`MultiDiGraph.add_edge` (u, v[, data])	Add a single directed edge to the digraph.
`MultiDiGraph.add_edges_from` (ebunch[, data])	Add all the edges in ebunch.
`MultiDiGraph.remove_edge` (u, v[, data])	Remove edge between (u,v).
`MultiDiGraph.remove_edges_from` (ebunch)	Remove all edges specified in ebunch.
`MultiDiGraph.add_star` (nlist[, data])	Add a star.
`MultiDiGraph.add_path` (nlist[, data])	Add a path.
`MultiDiGraph.add_cycle` (nlist[, data])	Add a cycle.
`MultiDiGraph.clear` ()	Remove all nodes and edges.

networkx.MultiDiGraph.add_node

add_node (*n*)

Add a single node n.

Parameters **n** : node

A node n can be any hashable Python object except None.
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.

Notes

On many platforms hashable items also include mutables such as DiGraphs, though one should be careful that the hash doesn't change on mutables.

Examples

```
>>> G=nx.DiGraph()
>>> G.add_node(1)
>>> G.add_node('Hello')
>>> K3=nx.complete_graph(3)
>>> G.add_node(K3)
>>> G.number_of_nodes()
3
```

networkx.MultiDiGraph.add_nodes_from

add_nodes_from(*nbunch*)

Add nodes from nbunch.

Parameters **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.

Examples

```
>>> G=nx.DiGraph()
>>> G.add_nodes_from('Hello')
>>> K3=nx.complete_graph(3)
>>> G.add_nodes_from(K3)
>>> sorted(G.nodes())
[0, 1, 2, 'H', 'e', 'l', 'o']
```

networkx.MultiDiGraph.remove_node

remove_node(*n*)

Remove node n.

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

Examples

```
>>> G=nx.complete_graph(3)  # complete graph on 3 nodes, K3
>>> G.edges()
[(0, 1), (0, 2), (1, 2)]
>>> G.remove_node(1)
>>> G.edges()
[(0, 2)]
```

networkx.MultiDiGraph.remove_nodes_from

remove_nodes_from(*nbunch*)

Remove nodes specified in nbunch.

Parameters **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.

Examples

```
>>> G=nx.complete_graph(3)  # complete graph on 3 nodes, K3
>>> e=G.nodes()
>>> e
[0, 1, 2]
>>> G.remove_nodes_from(e)
>>> G.nodes()
[]
```

networkx.MultiDiGraph.add_edge

add_edge(*u, v, data=1*)

Add a single directed edge to the digraph.

x is an arbitrary (not necessarily hashable) object associated with this edge. It can be used to associate one or more, labels, data records, weights or any arbirary objects to edges. The default is the Python None.

For example, after creation, the edge (1,2,"blue") can be added

```
>>> G=nx.MultiDiGraph()
>>> G.add_edge(1,2,"blue")
```

Two successive calls to G.add_edge(1,2,"red") will result in 2 edges of the form (1,2,"red") that can not be distinguished from one another.

networkx.MultiDiGraph.add_edges_from

add_edges_from(*ebunch, data=1*)

Add all the edges in ebunch.

Parameters **ebunch** : list or container of edges

The container must be iterable or an iterator. It is iterated over once. Adding the same edge twice has no effect and does not raise an exception. The edges in ebunch must be 2-tuples (u,v) or 3-tuples (u,v,d).

data : any Python object The default data for edges with no data given. If unspecified the integer 1 will be used.

See Also:

add_edge

Examples

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

networkx.MultiDiGraph.remove_edge

`remove_edge`(*u, v, data=None*)

Remove edge between (u,v).

If d is defined only remove the first edge found with edgedata == d.

If d is None, remove all edges between u and v.

networkx.MultiDiGraph.remove_edges_from

`remove_edges_from`(*ebunch*)

Remove all edges specified in ebunch.

Parameters **ebunch: list or container of edge tuples** :

A container of edge 2-tuples (u,v) or edge 3-tuples(u,v,d) though d is ignored unless we are a multigraph.

See Also:

`remove_edge`

Notes

Will fail silently if the edge (u,v) is not in the graph.

Examples

```
>>> G=nx.path_graph(4)
>>> ebunch=[(1,2),(2,3)]
>>> G.remove_edges_from(ebunch)
```

networkx.MultiDiGraph.add_star

`add_star`(*nlist, data=None*)

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.

data : list or iterable, optional

Data to add to the edges in the path. The length should be one less than len(nlist).

Examples

```
>>> G=nx.Graph()
>>> G.add_star([0,1,2,3])
>>> G.add_star([10,11,12],data=['a','b'])
```

networkx.MultiDiGraph.add_path

add_path(*nlist, data=None*)

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.

data : list or iterable, optional

Data to add to the edges in the path. The length should be one less than len(nlist).

Examples

```
>>> G=nx.Graph()
>>> G.add_path([0,1,2,3])
>>> G.add_path([10,11,12],data=['a','b'])
```

networkx.MultiDiGraph.add_cycle

add_cycle(*nlist, data=None*)

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.

data : list or iterable, optional

Data to add to the edges in the path. The length should be the same as nlist.

Examples

```
>>> G=nx.Graph()
>>> G.add_cycle([0,1,2,3])
>>> G.add_cycle([10,11,12],data=['a','b','c'])
```

networkx.MultiDiGraph.clear

clear()

Remove all nodes and edges.

This also removes the name.

Examples

```
>>> G=nx.path_graph(4)
>>> G.clear()
>>> G.nodes()
[]
>>> G.edges()
[]
```

Iterating over nodes and edges

MultiDiGraph.nodes ()	Return a list of the nodes.
MultiDiGraph.nodes_iter ()	Return an iterator for the nodes.
MultiDiGraph.__iter__ ()	Iterate over the nodes. Use "for n in G".
MultiDiGraph.edges ([nbunch, data])	Return a list of edges.
MultiDiGraph.edges_iter ([nbunch, data])	Return an iterator over the edges.
MultiDiGraph.get_edge (u, v[, no_edge])	Return a list of edge data for all edges between u and v.
MultiDiGraph.neighbors (n)	Return a list of the nodes connected to the node n.
MultiDiGraph.neighbors_iter (n)	Return an iterator over all neighbors of node n.
MultiDiGraph.__getitem__ (n)	Return the neighbors of node n. Use "G[n]".
MultiDiGraph.successors (n)	Return a list of the nodes connected to the node n.
MultiDiGraph.successors_iter (n)	Return an iterator over all neighbors of node 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 as a Python list of lists
MultiDiGraph.adjacency_iter ()	Return an iterator of (node, adjacency dict) tuples for all nodes.
MultiDiGraph.nbunch_iter ([nbunch])	Return an iterator of nodes contained in nbunch that are also in the graph.

networkx.MultiDiGraph.nodes

nodes()
Return a list of the nodes.

Examples

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

networkx.MultiDiGraph.nodes_iter
nodes_iter()
Return an iterator for the nodes.

Notes

It is simpler and equivalent to use the expression "for n in G"

```
>>> G=nx.path_graph(3)
>>> for n in G:
...     print n,
0 1 2
```

Examples

```
>>> G=nx.path_graph(3)
>>> for n in G.nodes_iter():
...     print n,
0 1 2
```

You can also say

```
>>> G=nx.path_graph(3)
>>> for n in G:
...     print n,
0 1 2
```

networkx.MultiDiGraph.__iter__
__iter__()
Iterate over the nodes. Use "for n in G".

Examples

```
>>> G=nx.path_graph(4)
>>> print [n for n in G]
[0, 1, 2, 3]
```

networkx.MultiDiGraph.edges

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

Return a list of edges.

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

data : bool

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

Returns **Edges that are adjacent to any node in nbunch,** :

or a list of all edges if nbunch is not specified. :

Examples

```
>>> G=nx.path_graph(4)
>>> G.edges()
[(0, 1), (1, 2), (2, 3)]
>>> G.edges(data=True) # default edge data is 1
[(0, 1, 1), (1, 2, 1), (2, 3, 1)]
```

networkx.MultiDiGraph.edges_iter

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

Return an iterator over the edges.

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

data : bool

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

Returns **An iterator over edges that are adjacent to any node in nbunch,** :

or over all edges if nbunch is not specified. :

Examples

```
>>> G=nx.path_graph(4)
>>> G.edges()
[(0, 1), (1, 2), (2, 3)]
>>> G.edges(data=True) # default edge data is 1
[(0, 1, 1), (1, 2, 1), (2, 3, 1)]
```

networkx.MultiDiGraph.get_edge

get_edge (*u, v, no_edge=None*)

Return a list of edge data for all edges between u and v.

If no_edge is specified and the edge (u,v) isn't found, (and u and v are nodes), return the value of no_edge. If no_edge is None (or u or v aren't nodes) raise an exception.

networkx.MultiDiGraph.neighbors

neighbors(*n*)

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

Notes

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

```
>>> G=nx.Graph()
>>> G.add_edge('a','b','data')
>>> G['a']
{'b': 'data'}
```

Examples

```
>>> G=nx.path_graph(4)
>>> G.neighbors(0)
[1]
```

networkx.MultiDiGraph.neighbors_iter

neighbors_iter(*n*)

Return an iterator over all neighbors of node n.

Notes

It is faster to iterate over the using the idiom >>> print [n for n in G[0]] [1]

Examples

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

networkx.MultiDiGraph.__getitem__

__getitem__(*n*)

Return the neighbors of node n. Use "G[n]".

Notes

G[n] is similar to G.neighbors(n) but the internal data dictionary is returned instead of a list.

G[u][v] returns the edge data for edge (u,v).

```
>>> G=nx.path_graph(4)
>>> print G[0][1]
1
```

Assigning G[u][v] may corrupt the graph data structure.

Examples

```
>>> G=nx.path_graph(4)
>>> print G[0]
{1: 1}
```

networkx.MultiDiGraph.successors

successors(*n*)

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

Notes

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

```
>>> G=nx.Graph()
>>> G.add_edge('a','b','data')
>>> G['a']
{'b': 'data'}
```

Examples

```
>>> G=nx.path_graph(4)
>>> G.neighbors(0)
[1]
```

networkx.MultiDiGraph.successors_iter

successors_iter(*n*)

Return an iterator over all neighbors of node n.

Notes

It is faster to iterate over the using the idiom >>> print [n for n in G[0]] [1]

Examples

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

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 as a Python list of lists

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

Examples

```
>>> G=nx.path_graph(4)
>>> G.adjacency_list() # in sorted node order 0,1,2,3
[[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.

Notes

The dictionary returned is part of the internal graph data structure; changing it could corrupt that structure. This is meant for fast inspection, not mutation.

For MultiGraph/MultiDiGraph multigraphs, a list of edge data is the value in the dict.

Examples

```
>>> G=nx.path_graph(4)
>>> [(n,nbrdict) for n,nbrdict in G.adjacency_iter()]
[(0, {1: 1}), (1, {0: 1, 2: 1}), (2, {1: 1, 3: 1}), (3, {2: 1})]
```

networkx.MultiDiGraph.nbunch_iter

nbunch_iter(*nbunch=None*)

Return an iterator of nodes contained in nbunch that are also in the graph.

Parameters **nbunch** : list, iterable

A container of nodes that will be iterated through once (thus it should be an iterator or be iterable). Each element of the container should be a valid node type: any hashable type except None. If nbunch is None, return all edges data in the graph. Nodes in nbunch that are not in the graph will be (quietly) ignored.

Notes

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 values returned by an iterator nbunch is not hashable, a NetworkXError is raised.

Information about graph structure

MultiDiGraph.has_node (n)	Return True if graph has node n.
MultiDiGraph.__contains__ (n)	Return True if n is a node, False otherwise. Use "n in G".
MultiDiGraph.has_edge (u, v)	Return True if graph contains the edge (u,v), False otherwise.
MultiDiGraph.has_neighbor (u, v)	Return True if node u has neighbor v.
MultiDiGraph.nodes_with_selfloops ()	Return a list of nodes with self loops.
MultiDiGraph.selfloop_edges ()	Return a list of selfloop edges with data (3-tuples).
MultiDiGraph.order ()	Return the number of nodes.
MultiDiGraph.number_of_nodes ()	Return the number of nodes.
MultiDiGraph.__len__ ()	Return the number of nodes. Use "len(G)".
MultiDiGraph.size ([weighted])	Return the number of edges.
MultiDiGraph.number_of_edges ([u, v])	Return the number of edges between two nodes.
MultiDiGraph.number_of_selfloops ()	Return the number of selfloop edges counting multiple 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 graph has node n.

Notes

It is more readable and simpler to use >>> 0 in G True

Examples

```
>>> G=nx.path_graph(4)
>>> print G.has_node(0)
True
```

networkx.MultiDiGraph.__contains__

__contains__(*n*)

Return True if n is a node, False otherwise. Use "n in G".

Examples

```
>>> G=nx.path_graph(4)
>>> print 1 in G
True
```

networkx.MultiDiGraph.has_edge

has_edge(*u, v*)

Return True if graph contains the edge (u,v), False otherwise.

See Also:

Graph.has_neighbor

Examples

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

```
>>> G=nx.path_graph(4)
>>> G.has_edge(0,1)  # called using two nodes
True
>>> e=(0,1)
>>> G.has_edge(*e)  #  e is a 2-tuple (u,v)
True
>>> e=(0,1,'data')
>>> G.has_edge(*e[:2])  # e is a 3-tuple (u,v,d)
True
```

The following syntax are all equivalent:

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

networkx.MultiDiGraph.has_neighbor

has_neighbor(*u, v*)

Return True if node u has neighbor v.

This returns True if there exists any edge (u,v,data) for some data.

See Also:

has_edge

Examples

```
>>> G=nx.path_graph(4)
>>> G.has_neighbor(0,1)
True
>>> G.has_edge(0,1) # same as has_neighbor
True
>>> 1 in G[0]  # this gives KeyError if u not in G
True
```

networkx.MultiDiGraph.nodes_with_selfloops

nodes_with_selfloops()

Return a list of nodes with self loops.

Examples

```
>>> G=nx.Graph()
>>> G.add_edge(1,1)
>>> G.add_edge(1,2)
>>> G.nodes_with_selfloops()
[1]
```

networkx.MultiDiGraph.selfloop_edges

selfloop_edges()

Return a list of selfloop edges with data (3-tuples).

networkx.MultiDiGraph.order

order()

Return the number of nodes.

See Also:

Graph.order, Graph.__len__

networkx.MultiDiGraph.number_of_nodes

number_of_nodes()

Return the number of nodes.

Notes

This is the same as

```
>>> len(G)
4
```

and

```
>>> G.order()
4
```

Examples

```
>>> G=nx.path_graph(4)
>>> print len(G)
4
```

networkx.MultiDiGraph.__len__

__len__()

Return the number of nodes. Use "len(G)".

Examples

```
>>> G=nx.path_graph(4)
>>> print len(G)
4
```

networkx.MultiDiGraph.size

size(*weighted=False*)

Return the number of edges.

Parameters **weighted** : bool, optional

If True return the sum of the edge weights.

See Also:

`Graph.number_of_edges`

Examples

```
>>> G=nx.path_graph(4)
>>> G.size()
3
```

```
>>> G=nx.Graph()
>>> G.add_edge('a','b',2)
>>> G.add_edge('b','c',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

If u and v are specified, return the number of edges between u and v. Otherwise return the total number of all edges.

Examples

```
>>> G=nx.path_graph(4)
>>> 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 counting multiple edges.

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** : 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.

with_labels : False | True

Return a list of degrees (False) or a dictionary of degrees keyed by node (True).

weighted : False | True

If the graph is weighted return the weighted degree (the sum of edge weights).

Examples

```
>>> G=nx.path_graph(4)
>>> 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 that node.

Parameters **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.

weighted : False | True
If the graph is weighted return the weighted degree (the sum of edge weights).

Examples

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

Notes

This makes a complete of the graph but does not make copies of any underlying node or edge data. The node and edge data in the copy still point to the same objects as in the original.

networkx.MultiDiGraph.to_undirected

`to_undirected`()
Return an undirected representation of the digraph.

A new graph is returned 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 they appear as a double edge in the new multigraph.

networkx.MultiDiGraph.subgraph

`subgraph`(*nbunch*, *copy=True*)
Return the subgraph induced on nodes in nbunch.

Parameters **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.

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.

3.2.3 Labeled Graphs

The LabeledGraph and LabeledDiGraph classes extend the basic graphs by allowing arbitrary label data to be assigned to nodes.

Labeled Graph

Overview

class **LabeledGraph** (*data=None, name='', weighted=True*)

Adding and Removing Nodes and Edges

LabeledGraph.add_node (n[, data])	
LabeledGraph.add_nodes_from (nbunch[, data])	
LabeledGraph.remove_node (n)	
LabeledGraph.remove_nodes_from (nbunch)	
LabeledGraph.add_edge (u, v[, data])	Add an edge between u and v with optional data.
LabeledGraph.add_edges_from (ebunch[, data])	Add all the edges in ebunch.
LabeledGraph.remove_edge (u, v)	Remove the edge between (u,v).
LabeledGraph.remove_edges_from (ebunch)	Remove all edges specified in ebunch.
LabeledGraph.add_star (nlist[, data])	Add a star.
LabeledGraph.add_path (nlist[, data])	Add a path.
LabeledGraph.add_cycle (nlist[, data])	Add a cycle.
LabeledGraph.clear ()	

networkx.LabeledGraph.add_node
add_node (*n, data=None*)

networkx.LabeledGraph.add_nodes_from

add_nodes_from(*nbunch, data=None*)

networkx.LabeledGraph.remove_node

remove_node(*n*)

networkx.LabeledGraph.remove_nodes_from

remove_nodes_from(*nbunch*)

networkx.LabeledGraph.add_edge

add_edge(*u, v, data=1*)

Add an edge between u and v with optional data.

The nodes u and v will be automatically added if they are not already in the graph.

Parameters u,v : nodes

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

data : Python object

Edge data (or labels or objects) can be entered via the optional argument data which defaults to 1.

Some NetworkX algorithms are designed for weighted graphs for which the edge data must be a number. These may behave unpredictably for edge data that isn't a number.

See Also:

`Parallel`

Notes

Adding an edge that already exists *overwrites* the edgedata.

Examples

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

```
>>> G=nx.Graph()
>>> 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 the data myedge to the edge (1,2).

```
>>> myedge=1.3
>>> G.add_edge(1, 2, myedge)
```

networkx.LabeledGraph.add_edges_from

add_edges_from (*ebunch, data=1*)

Add all the edges in ebunch.

Parameters **ebunch** : list or container of edges

The container must be iterable or an iterator. It is iterated over once. Adding the same edge twice has no effect and does not raise an exception. The edges in ebunch must be 2-tuples (u,v) or 3-tuples (u,v,d).

data : any Python object The default data for edges with no data given. If unspecified the integer 1 will be used.

See Also:

add_edge

Examples

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

networkx.LabeledGraph.remove_edge

remove_edge (*u, v*)

Remove the edge between (u,v).

Parameters **u,v: nodes** :

See Also:

remove_edges_from remove a collection of edges

Examples

```
>>> G=nx.path_graph(4)
>>> G.remove_edge(0,1)
>>> e=(1,2)
>>> G.remove_edge(*e) # unpacks e from an edge tuple
>>> e=(2,3,'data')
>>> G.remove_edge(*e[:2]) # edge tuple with data
```

networkx.LabeledGraph.remove_edges_from

remove_edges_from (*ebunch*)

Remove all edges specified in ebunch.

Parameters **ebunch: list or container of edge tuples** :

A container of edge 2-tuples (u,v) or edge 3-tuples(u,v,d) though d is ignored unless we are a multigraph.

See Also:

remove_edge

Notes

Will fail silently if the edge (u,v) is not in the graph.

Examples

```
>>> G=nx.path_graph(4)
>>> ebunch=[(1,2),(2,3)]
>>> G.remove_edges_from(ebunch)
```

networkx.LabeledGraph.add_star

add_star(*nlist, data=None*)

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.

data : list or iterable, optional

Data to add to the edges in the path. The length should be one less than len(nlist).

Examples

```
>>> G=nx.Graph()
>>> G.add_star([0,1,2,3])
>>> G.add_star([10,11,12],data=['a','b'])
```

networkx.LabeledGraph.add_path

add_path(*nlist, data=None*)

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.

data : list or iterable, optional

Data to add to the edges in the path. The length should be one less than len(nlist).

Examples

```
>>> G=nx.Graph()
>>> G.add_path([0,1,2,3])
>>> G.add_path([10,11,12],data=['a','b'])
```

networkx.LabeledGraph.add_cycle

add_cycle(*nlist, data=None*)

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.

data : list or iterable, optional

Data to add to the edges in the path. The length should be the same as nlist.

Examples

```
>>> G=nx.Graph()
>>> G.add_cycle([0,1,2,3])
>>> G.add_cycle([10,11,12],data=['a','b','c'])
```

networkx.LabeledGraph.clear

clear()

Iterating over nodes and edges

LabeledGraph.nodes ([nbunch, data])	
LabeledGraph.nodes_iter ([nbunch, data])	
LabeledGraph.__iter__ ()	Iterate over the nodes. Use "for n in G".
LabeledGraph.edges ([nbunch, data])	Return a list of edges.
LabeledGraph.edges_iter ([nbunch, data])	Return an iterator over the edges.
LabeledGraph.get_edge (u, v[, default])	Return the data associated with the edge (u,v).
LabeledGraph.neighbors (n)	Return a list of the nodes connected to the node n.
LabeledGraph.neighbors_iter (n)	Return an iterator over all neighbors of node n.
LabeledGraph.__getitem__ (n)	Return the neighbors of node n. Use "G[n]".
LabeledGraph.adjacency_list ()	Return an adjacency list as a Python list of lists
LabeledGraph.adjacency_iter ()	Return an iterator of (node, adjacency dict) tuples for all nodes.
LabeledGraph.nbunch_iter ([nbunch])	Return an iterator of nodes contained in nbunch that are also in the graph.

networkx.LabeledGraph.nodes
nodes (*nbunch=None, data=False*)

networkx.LabeledGraph.nodes_iter
nodes_iter (*nbunch=None, data=False*)

networkx.LabeledGraph.__iter__
__iter__ ()

Iterate over the nodes. Use "for n in G".

Examples

```
>>> G=nx.path_graph(4)
>>> print [n for n in G]
[0, 1, 2, 3]
```

networkx.LabeledGraph.edges

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

Return a list of edges.

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

data : bool

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

Returns **Edges that are adjacent to any node in nbunch,** :

or a list of all edges if nbunch is not specified. :

Examples

```
>>> G=nx.path_graph(4)
>>> G.edges()
[(0, 1), (1, 2), (2, 3)]
>>> G.edges(data=True) # default edge data is 1
[(0, 1, 1), (1, 2, 1), (2, 3, 1)]
```

networkx.LabeledGraph.edges_iter

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

Return an iterator over the edges.

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

data : bool

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

Returns **An iterator over edges that are adjacent to any node in nbunch,** :

or over all edges if nbunch is not specified. :

Examples

```
>>> G=nx.path_graph(4)
>>> G.edges()
[(0, 1), (1, 2), (2, 3)]
>>> G.edges(data=True) # default edge data is 1
[(0, 1, 1), (1, 2, 1), (2, 3, 1)]
```

networkx.LabeledGraph.get_edge

get_edge (*u, v, default=None*)

Return the data associated with the edge (u,v).

Parameters **u,v** : nodes

If u or v are not nodes in graph an exception is raised.

default: any Python object :
Value to return if the edge (u,v) is not found. If not specified, raise an exception.

Notes

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

```
>>> G[0][1]
1
```

Examples

```
>>> G=nx.path_graph(4) # path graph with edge data all set to 1
>>> G.get_edge(0,1)
1
>>> e=(0,1)
>>> G.get_edge(*e) # tuple form
1
```

networkx.LabeledGraph.neighbors

neighbors (*n*)
Return a list of the nodes connected to the node n.

Notes

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

```
>>> G=nx.Graph()
>>> G.add_edge('a','b','data')
>>> G['a']
{'b': 'data'}
```

Examples

```
>>> G=nx.path_graph(4)
>>> G.neighbors(0)
[1]
```

networkx.LabeledGraph.neighbors_iter

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

Notes

It is faster to iterate over the using the idiom >>> print [n for n in G[0]] [1]

Examples

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

networkx.LabeledGraph.__getitem__

__getitem__(*n*)

Return the neighbors of node n. Use "G[n]".

Notes

G[n] is similar to G.neighbors(n) but the internal data dictionary is returned instead of a list.

G[u][v] returns the edge data for edge (u,v).

```
>>> G=nx.path_graph(4)
>>> print G[0][1]
1
```

Assigning G[u][v] may corrupt the graph data structure.

Examples

```
>>> G=nx.path_graph(4)
>>> print G[0]
{1: 1}
```

networkx.LabeledGraph.adjacency_list

adjacency_list()

Return an adjacency list as a Python list of lists

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

Examples

```
>>> G=nx.path_graph(4)
>>> G.adjacency_list() # in sorted node order 0,1,2,3
[[1], [0, 2], [1, 3], [2]]
```

networkx.LabeledGraph.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.

Notes

The dictionary returned is part of the internal graph data structure; changing it could corrupt that structure. This is meant for fast inspection, not mutation.

For MultiGraph/MultiDiGraph multigraphs, a list of edge data is the value in the dict.

Examples

```
>>> G=nx.path_graph(4)
>>> [(n,nbrdict) for n,nbrdict in G.adjacency_iter()]
[(0, {1: 1}), (1, {0: 1, 2: 1}), (2, {1: 1, 3: 1}), (3, {2: 1})]
```

networkx.LabeledGraph.nbunch_iter

nbunch_iter(*nbunch=None*)

Return an iterator of nodes contained in nbunch that are also in the graph.

Parameters **nbunch** : list, iterable

A container of nodes that will be iterated through once (thus it should be an iterator or be iterable). Each element of the container should be a valid node type: any hashable type except None. If nbunch is None, return all edges data in the graph. Nodes in nbunch that are not in the graph will be (quietly) ignored.

Notes

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 values returned by an iterator nbunch is not hashable, a NetworkXError is raised.

Information about graph structure

`LabeledGraph.has_node` (n)	Return True if graph has node n.
`LabeledGraph.__contains__` (n)	Return True if n is a node, False otherwise. Use "n in G".
`LabeledGraph.has_edge` (u, v)	Return True if graph contains the edge (u,v), False otherwise.
`LabeledGraph.has_neighbor` (u, v)	Return True if node u has neighbor v.
`LabeledGraph.nodes_with_selfloops` ()	Return a list of nodes with self loops.
`LabeledGraph.selfloop_edges` ([data])	Return a list of selfloop edges
`LabeledGraph.order` ()	Return the number of nodes.
`LabeledGraph.number_of_nodes` ()	Return the number of nodes.
`LabeledGraph.__len__` ()	Return the number of nodes. Use "len(G)".
`LabeledGraph.size` ([weighted])	Return the number of edges.
`LabeledGraph.number_of_edges` ([u, v])	Return the number of edges between two nodes.
`LabeledGraph.number_of_selfloops` ()	Return the number of selfloop edges (edge from a node to itself).
`LabeledGraph.degree` ([nbunch, with_labels, ...])	Return the degree of a node or nodes.
`LabeledGraph.degree_iter` ([nbunch, weighted])	Return an iterator for (node, degree).

networkx.LabeledGraph.has_node

has_node (*n*)

Return True if graph has node n.

Notes

It is more readable and simpler to use >>> 0 in G True

Examples

```
>>> G=nx.path_graph(4)
>>> print G.has_node(0)
True
```

networkx.LabeledGraph.__contains__

__contains__(*n*)

Return True if n is a node, False otherwise. Use "n in G".

Examples

```
>>> G=nx.path_graph(4)
>>> print 1 in G
True
```

networkx.LabeledGraph.has_edge

has_edge(*u, v*)

Return True if graph contains the edge (u,v), False otherwise.

See Also:

`Graph.has_neighbor`

Examples

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

```
>>> G=nx.path_graph(4)
>>> G.has_edge(0,1)  # called using two nodes
True
>>> e=(0,1)
>>> G.has_edge(*e)  #  e is a 2-tuple (u,v)
True
>>> e=(0,1,'data')
>>> G.has_edge(*e[:2])  # e is a 3-tuple (u,v,d)
True
```

The following syntax are all equivalent:

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

networkx.LabeledGraph.has_neighbor

has_neighbor(*u, v*)

Return True if node u has neighbor v.

This returns True if there exists any edge (u,v,data) for some data.

See Also:

`has_edge`

Examples

```
>>> G=nx.path_graph(4)
>>> G.has_neighbor(0,1)
True
>>> G.has_edge(0,1) # same as has_neighbor
True
>>> 1 in G[0]  # this gives KeyError if u not in G
True
```

networkx.LabeledGraph.nodes_with_selfloops

nodes_with_selfloops()

Return a list of nodes with self loops.

Examples

```
>>> G=nx.Graph()
>>> G.add_edge(1,1)
>>> G.add_edge(1,2)
>>> G.nodes_with_selfloops()
[1]
```

networkx.LabeledGraph.selfloop_edges

selfloop_edges(*data=False*)

Return a list of selfloop edges

Parameters **data** : bool

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

Examples

```
>>> G=nx.Graph()
>>> G.add_edge(1,1)
>>> G.add_edge(1,2)
>>> G.selfloop_edges()
[(1, 1)]
>>> G.selfloop_edges(data=True)
[(1, 1, 1)]
```

networkx.LabeledGraph.order

order()

Return the number of nodes.

See Also:

Graph.order, Graph.__len__

networkx.LabeledGraph.number_of_nodes

number_of_nodes()

Return the number of nodes.

Notes

This is the same as

```
>>> len(G)
4
```

and

```
>>> G.order()
4
```

Examples

```
>>> G=nx.path_graph(4)
>>> print len(G)
4
```

networkx.LabeledGraph.__len__

__len__()

Return the number of nodes. Use "len(G)".

Examples

```
>>> G=nx.path_graph(4)
>>> print len(G)
4
```

networkx.LabeledGraph.size

size(*weighted=False*)

Return the number of edges.

Parameters **weighted** : bool, optional

If True return the sum of the edge weights.

See Also:

Graph.number_of_edges

Examples

```
>>> G=nx.path_graph(4)
>>> G.size()
3
```

```
>>> G=nx.Graph()
>>> G.add_edge('a','b',2)
>>> G.add_edge('b','c',4)
>>> G.size()
2
```

```
>>> G.size(weighted=True)
6
```

networkx.LabeledGraph.number_of_edges

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

Return the number of edges between two nodes.

Parameters u,v : nodes

If u and v are specified, return the number of edges between u and v. Otherwise return the total number of all edges.

Examples

```
>>> G=nx.path_graph(4)
>>> G.number_of_edges()
3
>>> G.number_of_edges(0,1)
1
>>> e=(0,1)
>>> G.number_of_edges(*e)
1
```

networkx.LabeledGraph.number_of_selfloops

number_of_selfloops ()

Return the number of selfloop edges (edge from a node to itself).

Examples

```
>>> G=nx.Graph()
>>> G.add_edge(1,1)
>>> G.add_edge(1,2)
>>> G.number_of_selfloops()
1
```

networkx.LabeledGraph.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 : 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.

with_labels : False | True

Return a list of degrees (False) or a dictionary of degrees keyed by node (True).

weighted : False | True

If the graph is weighted return the weighted degree (the sum of edge weights).

Examples

```
>>> G=nx.path_graph(4)
>>> G.degree(0)
1
>>> G.degree([0,1])
[1, 2]
>>> G.degree([0,1],with_labels=True)
{0: 1, 1: 2}
```

networkx.LabeledGraph.degree_iter

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

Return an iterator for (node, degree).

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

Parameters **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.

weighted : False | True

If the graph is weighted return the weighted degree (the sum of edge weights).

Examples

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

LabeledGraph.copy ()	Return a copy of the graph.
LabeledGraph.to_directed ()	
LabeledGraph.subgraph (nbunch[, copy])	

networkx.LabeledGraph.copy

copy()

Return a copy of the graph.

Notes

This makes a complete of the graph but does not make copies of any underlying node or edge data. The node and edge data in the copy still point to the same objects as in the original.

networkx.LabeledGraph.to_directed
to_directed()

networkx.LabeledGraph.subgraph
subgraph(*nbunch, copy=True*)

Labeled DiGraph

Overview

class **LabeledDiGraph**(*data=None, name='', weighted=True*)

Adding and Removing Nodes and Edges

LabeledDiGraph.add_node(n[, data])	
LabeledDiGraph.add_nodes_from(nbunch[, data])	
LabeledDiGraph.remove_node(n)	
LabeledDiGraph.remove_nodes_from(nbunch)	
LabeledDiGraph.add_edge(u, v[, data])	Add an edge between u and v with optional data.
LabeledDiGraph.add_edges_from(ebunch[, data])	Add all the edges in ebunch.
LabeledDiGraph.remove_edge(u, v)	Remove the edge between (u,v).
LabeledDiGraph.remove_edges_from(ebunch)	Remove all edges specified in ebunch.
LabeledDiGraph.add_star(nlist[, data])	Add a star.
LabeledDiGraph.add_path(nlist[, data])	Add a path.
LabeledDiGraph.add_cycle(nlist[, data])	Add a cycle.
LabeledDiGraph.clear()	

networkx.LabeledDiGraph.add_node
add_node(*n, data=None*)

networkx.LabeledDiGraph.add_nodes_from
add_nodes_from(*nbunch, data=None*)

networkx.LabeledDiGraph.remove_node
remove_node(*n*)

networkx.LabeledDiGraph.remove_nodes_from

remove_nodes_from(*nbunch*)

networkx.LabeledDiGraph.add_edge

add_edge(*u, v, data=1*)

Add an edge between u and v with optional data.

The nodes u and v will be automatically added if they are not already in the graph.

Parameters u,v : nodes

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

data : Python object

Edge data (or labels or objects) can be entered via the optional argument data which defaults to 1.

Some NetworkX algorithms are designed for weighted graphs for which the edge data must be a number. These may behave unpredictably for edge data that isn't a number.

See Also:

`Parallel`

Notes

Adding an edge that already exists *overwrites* the edgedata.

Examples

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

```
>>> G=nx.DiGraph()
>>> 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 the data myedge to the edge (1,2).

```
>>> myedge=1.3
>>> G.add_edge(1, 2, myedge)
```

networkx.LabeledDiGraph.add_edges_from

add_edges_from(*ebunch, data=1*)

Add all the edges in ebunch.

Parameters ebunch : list or container of edges

The container must be iterable or an iterator. It is iterated over once. Adding the same edge twice has no effect and does not raise an exception. The edges in ebunch must be 2-tuples (u,v) or 3-tuples (u,v,d).

data : any Python object The default data for edges with no data given. If unspecified the integer 1 will be used.

See Also:

`add_edge`

Examples

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

networkx.LabeledDiGraph.remove_edge

remove_edge (*u, v*)

Remove the edge between (u,v).

Parameters **u,v: nodes** :

See Also:

remove_edges_from remove a collection of edges

Examples

```
>>> G=nx.path_graph(4)
>>> G.remove_edge(0,1)
>>> e=(1,2)
>>> G.remove_edge(*e) # unpacks e from an edge tuple
>>> e=(2,3,'data')
>>> G.remove_edge(*e[:2]) # edge tuple with data
```

networkx.LabeledDiGraph.remove_edges_from

remove_edges_from (*ebunch*)

Remove all edges specified in ebunch.

Parameters **ebunch: list or container of edge tuples** :

A container of edge 2-tuples (u,v) or edge 3-tuples(u,v,d) though d is ignored unless we are a multigraph.

See Also:

remove_edge

Notes

Will fail silently if the edge (u,v) is not in the graph.

Examples

```
>>> G=nx.path_graph(4)
>>> ebunch=[(1,2),(2,3)]
>>> G.remove_edges_from(ebunch)
```

networkx.LabeledDiGraph.add_star

add_star(*nlist, data=None*)

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.

data : list or iterable, optional

Data to add to the edges in the path. The length should be one less than len(nlist).

Examples

```
>>> G=nx.Graph()
>>> G.add_star([0,1,2,3])
>>> G.add_star([10,11,12],data=['a','b'])
```

networkx.LabeledDiGraph.add_path

add_path(*nlist, data=None*)

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.

data : list or iterable, optional

Data to add to the edges in the path. The length should be one less than len(nlist).

Examples

```
>>> G=nx.Graph()
>>> G.add_path([0,1,2,3])
>>> G.add_path([10,11,12],data=['a','b'])
```

networkx.LabeledDiGraph.add_cycle

add_cycle(*nlist, data=None*)

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.

data : list or iterable, optional

Data to add to the edges in the path. The length should be the same as nlist.

Examples

```
>>> G=nx.Graph()
>>> G.add_cycle([0,1,2,3])
>>> G.add_cycle([10,11,12],data=['a','b','c'])
```

networkx.LabeledDiGraph.clear

clear()

Iterating over nodes and edges

LabeledDiGraph.nodes ([nbunch, data])	
LabeledDiGraph.nodes_iter ([nbunch, data])	
LabeledDiGraph.__iter__ ()	Iterate over the nodes. Use "for n in G".
LabeledDiGraph.edges ([nbunch, data])	Return a list of edges.
LabeledDiGraph.edges_iter ([nbunch, data])	Return an iterator over the edges.
LabeledDiGraph.get_edge (u, v[, default])	Return the data associated with the edge (u,v).
LabeledDiGraph.neighbors (n)	Return a list of the nodes connected to the node n.
LabeledDiGraph.neighbors_iter (n)	Return an iterator over all neighbors of node n.
LabeledDiGraph.__getitem__ (n)	Return the neighbors of node n. Use "G[n]".
LabeledDiGraph.successors (n)	Return a list of the nodes connected to the node n.
LabeledDiGraph.successors_iter (n)	Return an iterator over all neighbors of node n.
LabeledDiGraph.predecessors (n)	Return a list of predecessor nodes of n.
LabeledDiGraph.predecessors_iter (n)	Return an iterator over predecessor nodes of n.
LabeledDiGraph.adjacency_list ()	Return an adjacency list as a Python list of lists
LabeledDiGraph.adjacency_iter ()	Return an iterator of (node, adjacency dict) tuples for all nodes.
LabeledDiGraph.nbunch_iter ([nbunch])	Return an iterator of nodes contained in nbunch that are also in the graph.

networkx.LabeledDiGraph.nodes
nodes (*nbunch=None, data=False*)

networkx.LabeledDiGraph.nodes_iter
nodes_iter (*nbunch=None, data=False*)

networkx.LabeledDiGraph.__iter__

__iter__()

Iterate over the nodes. Use "for n in G".

Examples

```
>>> G=nx.path_graph(4)
>>> print [n for n in G]
[0, 1, 2, 3]
```

networkx.LabeledDiGraph.edges

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

Return a list of edges.

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

data : bool

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

Returns Edges that are adjacent to any node in nbunch, :

or a list of all edges if nbunch is not specified. :

Examples

```
>>> G=nx.path_graph(4)
>>> G.edges()
[(0, 1), (1, 2), (2, 3)]
>>> G.edges(data=True) # default edge data is 1
[(0, 1, 1), (1, 2, 1), (2, 3, 1)]
```

networkx.LabeledDiGraph.edges_iter

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

Return an iterator over the edges.

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

data : bool

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

Returns An iterator over edges that are adjacent to any node in nbunch, :

or over all edges if nbunch is not specified. :

Examples

```
>>> G=nx.path_graph(4)
>>> G.edges()
[(0, 1), (1, 2), (2, 3)]
>>> G.edges(data=True) # default edge data is 1
[(0, 1, 1), (1, 2, 1), (2, 3, 1)]
```

networkx.LabeledDiGraph.get_edge

get_edge (*u, v, default=None*)

Return the data associated with the edge (u,v).

Parameters u,v : nodes

If u or v are not nodes in graph an exception is raised.

default: any Python object :

Value to return if the edge (u,v) is not found. If not specified, raise an exception.

Notes

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

```
>>> G[0][1]
1
```

Examples

```
>>> G=nx.path_graph(4) # path graph with edge data all set to 1
>>> G.get_edge(0,1)
1
>>> e=(0,1)
>>> G.get_edge(*e) # tuple form
1
```

networkx.LabeledDiGraph.neighbors

neighbors (*n*)

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

Notes

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

```
>>> G=nx.Graph()
>>> G.add_edge('a','b','data')
>>> G['a']
{'b': 'data'}
```

Examples

```
>>> G=nx.path_graph(4)
>>> G.neighbors(0)
[1]
```

networkx.LabeledDiGraph.neighbors_iter

neighbors_iter(*n*)

Return an iterator over all neighbors of node n.

Notes

It is faster to iterate over the using the idiom >>> print [n for n in G[0]] [1]

Examples

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

networkx.LabeledDiGraph.__getitem__

__getitem__(*n*)

Return the neighbors of node n. Use "G[n]".

Notes

G[n] is similar to G.neighbors(n) but the internal data dictionary is returned instead of a list.

G[u][v] returns the edge data for edge (u,v).

```
>>> G=nx.path_graph(4)
>>> print G[0][1]
1
```

Assigning G[u][v] may corrupt the graph data structure.

Examples

```
>>> G=nx.path_graph(4)
>>> print G[0]
{1: 1}
```

networkx.LabeledDiGraph.successors

successors(*n*)

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

Notes

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

```
>>> G=nx.Graph()
>>> G.add_edge('a','b','data')
>>> G['a']
{'b': 'data'}
```

Examples

```
>>> G=nx.path_graph(4)
>>> G.neighbors(0)
[1]
```

networkx.LabeledDiGraph.successors_iter

successors_iter(*n*)

Return an iterator over all neighbors of node n.

Notes

It is faster to iterate over the using the idiom >>> print [n for n in G[0]] [1]

Examples

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

networkx.LabeledDiGraph.predecessors

predecessors(*n*)

Return a list of predecessor nodes of n.

networkx.LabeledDiGraph.predecessors_iter

predecessors_iter(*n*)

Return an iterator over predecessor nodes of n.

networkx.LabeledDiGraph.adjacency_list

adjacency_list()

Return an adjacency list as a Python list of lists

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

Examples

```
>>> G=nx.path_graph(4)
>>> G.adjacency_list() # in sorted node order 0,1,2,3
[[1], [0, 2], [1, 3], [2]]
```

networkx.LabeledDiGraph.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.

Notes

The dictionary returned is part of the internal graph data structure; changing it could corrupt that structure. This is meant for fast inspection, not mutation.

For MultiGraph/MultiDiGraph multigraphs, a list of edge data is the value in the dict.

Examples

```
>>> G=nx.path_graph(4)
>>> [(n,nbrdict) for n,nbrdict in G.adjacency_iter()]
[(0, {1: 1}), (1, {0: 1, 2: 1}), (2, {1: 1, 3: 1}), (3, {2: 1})]
```

networkx.LabeledDiGraph.nbunch_iter

nbunch_iter(*nbunch=None*)

Return an iterator of nodes contained in nbunch that are also in the graph.

Parameters **nbunch** : list, iterable

A container of nodes that will be iterated through once (thus it should be an iterator or be iterable). Each element of the container should be a valid node type: any hashable type except None. If nbunch is None, return all edges data in the graph. Nodes in nbunch that are not in the graph will be (quietly) ignored.

Notes

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 values returned by an iterator nbunch is not hashable, a NetworkXError is raised.

Information about graph structure

`LabeledDiGraph.has_node` (n)	Return True if graph has node n.
`LabeledDiGraph.__contains__` (n)	Return True if n is a node, False otherwise. Use "n in G".
`LabeledDiGraph.has_edge` (u, v)	Return True if graph contains the edge (u,v), False otherwise.
`LabeledDiGraph.has_neighbor` (u, v)	Return True if node u has neighbor v.
`LabeledDiGraph.nodes_with_selfloops` ()	Return a list of nodes with self loops.
`LabeledDiGraph.selfloop_edges` ([data])	Return a list of selfloop edges
`LabeledDiGraph.order` ()	Return the number of nodes.
`LabeledDiGraph.number_of_nodes` ()	Return the number of nodes.
`LabeledDiGraph.__len__` ()	Return the number of nodes. Use "len(G)".
`LabeledDiGraph.size` ([weighted])	Return the number of edges.
`LabeledDiGraph.number_of_edges` ([u, v])	Return the number of edges between two nodes.
`LabeledDiGraph.number_of_selfloops` ()	Return the number of selfloop edges (edge from a node to itself).
`LabeledDiGraph.degree` ([nbunch, with_labels, ...])	Return the degree of a node or nodes.
`LabeledDiGraph.degree_iter` ([nbunch, weighted])	Return an iterator for (node, degree).

networkx.LabeledDiGraph.has_node

has_node (*n*)

Return True if graph has node n.

Notes

It is more readable and simpler to use >>> 0 in G True

Examples

```
>>> G=nx.path_graph(4)
>>> print G.has_node(0)
True
```

networkx.LabeledDiGraph.__contains__

__contains__(*n*)

Return True if n is a node, False otherwise. Use "n in G".

Examples

```
>>> G=nx.path_graph(4)
>>> print 1 in G
True
```

networkx.LabeledDiGraph.has_edge

has_edge(*u, v*)

Return True if graph contains the edge (u,v), False otherwise.

See Also:

Graph.has_neighbor

Examples

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

```
>>> G=nx.path_graph(4)
>>> G.has_edge(0,1)  # called using two nodes
True
>>> e=(0,1)
>>> G.has_edge(*e)  #  e is a 2-tuple (u,v)
True
>>> e=(0,1,'data')
>>> G.has_edge(*e[:2])  # e is a 3-tuple (u,v,d)
True
```

The following syntax are all equivalent:

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

networkx.LabeledDiGraph.has_neighbor

has_neighbor(*u, v*)

Return True if node u has neighbor v.

This returns True if there exists any edge (u,v,data) for some data.

See Also:

has_edge

Examples

```
>>> G=nx.path_graph(4)
>>> G.has_neighbor(0,1)
True
>>> G.has_edge(0,1) # same as has_neighbor
True
>>> 1 in G[0]  # this gives KeyError if u not in G
True
```

networkx.LabeledDiGraph.nodes_with_selfloops

nodes_with_selfloops()

Return a list of nodes with self loops.

Examples

```
>>> G=nx.Graph()
>>> G.add_edge(1,1)
>>> G.add_edge(1,2)
>>> G.nodes_with_selfloops()
[1]
```

networkx.LabeledDiGraph.selfloop_edges

selfloop_edges(*data=False*)

Return a list of selfloop edges

Parameters **data** : bool

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

Examples

```
>>> G=nx.Graph()
>>> G.add_edge(1,1)
>>> G.add_edge(1,2)
>>> G.selfloop_edges()
[(1, 1)]
>>> G.selfloop_edges(data=True)
[(1, 1, 1)]
```

networkx.LabeledDiGraph.order

order()

Return the number of nodes.

See Also:

Graph.order, Graph.__len__

networkx.LabeledDiGraph.number_of_nodes

number_of_nodes()

Return the number of nodes.

Notes

This is the same as

```
>>> len(G)
4
```

and

```
>>> G.order()
4
```

Examples

```
>>> G=nx.path_graph(4)
>>> print len(G)
4
```

networkx.LabeledDiGraph.__len__

__len__()

Return the number of nodes. Use "len(G)".

Examples

```
>>> G=nx.path_graph(4)
>>> print len(G)
4
```

networkx.LabeledDiGraph.size

size(*weighted=False*)

Return the number of edges.

Parameters **weighted** : bool, optional

If True return the sum of the edge weights.

See Also:

`Graph.number_of_edges`

Examples

```
>>> G=nx.path_graph(4)
>>> G.size()
3
```

```
>>> G=nx.Graph()
>>> G.add_edge('a','b',2)
>>> G.add_edge('b','c',4)
>>> G.size()
2
```

```
>>> G.size(weighted=True)
6
```

networkx.LabeledDiGraph.number_of_edges

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

Return the number of edges between two nodes.

Parameters u,v : nodes

If u and v are specified, return the number of edges between u and v. Otherwise return the total number of all edges.

Examples

```
>>> G=nx.path_graph(4)
>>> G.number_of_edges()
3
>>> G.number_of_edges(0,1)
1
>>> e=(0,1)
>>> G.number_of_edges(*e)
1
```

networkx.LabeledDiGraph.number_of_selfloops

number_of_selfloops()

Return the number of selfloop edges (edge from a node to itself).

Examples

```
>>> G=nx.Graph()
>>> G.add_edge(1,1)
>>> G.add_edge(1,2)
>>> G.number_of_selfloops()
1
```

networkx.LabeledDiGraph.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 : 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.

with_labels : False | True

Return a list of degrees (False) or a dictionary of degrees keyed by node (True).

weighted : False | True

If the graph is weighted return the weighted degree (the sum of edge weights).

Examples

```
>>> G=nx.path_graph(4)
>>> G.degree(0)
1
>>> G.degree([0,1])
[1, 2]
>>> G.degree([0,1],with_labels=True)
{0: 1, 1: 2}
```

networkx.LabeledDiGraph.degree_iter

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

Return an iterator for (node, degree).

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

Parameters **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.

weighted : False | True

If the graph is weighted return the weighted degree (the sum of edge weights).

Examples

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

LabeledDiGraph.copy ()	Return a copy of the graph.
LabeledDiGraph.to_undirected ()	Return an undirected representation of the digraph.
LabeledDiGraph.subgraph (nbunch[, copy])	
LabeledDiGraph.reverse ([copy])	Return the reverse of the graph

networkx.LabeledDiGraph.copy

copy ()

Return a copy of the graph.

Notes

This makes a complete of the graph but does not make copies of any underlying node or edge data. The node and edge data in the copy still point to the same objects as in the original.

networkx.LabeledDiGraph.to_undirected

to_undirected()

Return an undirected representation of the digraph.

A new graph is returned 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.

networkx.LabeledDiGraph.subgraph

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

networkx.LabeledDiGraph.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.

3.3 Algorithms

3.3.1 Boundary

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.

3.3.2 Centrality

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

networkx.betweenness_centrality

betweenness_centrality (*G, normalized=True, weighted_edges=False*)

Compute betweenness centrality for nodes.

Betweenness centrality is the fraction of number of shortests paths that pass through each node.

The keyword normalized (default=True) specifies whether the betweenness values are normalized by b=b/(n-1)(n-2) where n is the number of nodes in G.

The keyword weighted_edges (default=False) specifies whether to use edge weights (otherwise weights are all assumed equal).

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).

weighted_edges:: consider edge weights by running Dijkstra's algorithm (no effect on unweighted graphs).

sources:: list of nodes to consider as subgraph

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 fraction of number of shortests paths that go through each node counted according to the algorithm in Scientific collaboration networks: II. Shortest paths, weighted networks, and centrality, M. E. J. Newman, Phys. Rev. E 64, 016132 (2001).

This actually computes 'load' which is slightly diferent than betweenness.

Returns a dictionary of betweenness values keyed by node. The betweenness is normalized to be between [0,1].

If normalized=False the resulting betweenness is not normalized.

If weighted_edges is True then use Dijkstra for finding shortest paths.

networkx.edge_betweenness

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

Compute betweenness centrality for edges.

weighted_edges:: consider edge weights by running Dijkstra's algorithm (no effect on unweighted graphs).

sources:: list of nodes to consider as subgraph

networkx.degree_centrality

degree_centrality (*G, v=None*)

Compute degree centrality for nodes.

The degree centrality for a node v is the fraction of nodes it is connected to.

If v=None, returns a dict of degree centrality values keyed by node. Otherwise, returns the degree centrality of the node v.

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. Returns a dictionary of closeness centrality values keyed by node. The closeness centrality is normalized to be between 0 and 1.

3.3.3 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 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_using, ...])	Create a bipartite clique graph from a graph G.
`graph_clique_number` (G[, cliques])	Return the clique number (size the largest clique) for G. Optional list of cliques can be input if already computed.
`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 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 algorithm produces the list of maximal cliques each of which are a list of the members of the clique.

Based on Algol algorithm published by Bron & Kerbosch A C version is available as part of the rambin package. http://www.ram.org/computing/rambin/rambin.html

Reference:

```
@article{362367,
 author = {Coen Bron and Joep Kerbosch},
 title = {Algorithm 457: finding all cliques of an undirected graph},
 journal = {Commun. ACM},
 volume = {16},
 number = {9},
 year = {1973},
 issn = {0001-0782},
 pages = {575--577},
 doi = {http://doi.acm.org/10.1145/362342.362367},
 publisher = {ACM Press},
 }
```

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 B.node_type which is "Bottom" or "Top" appropriately.

if fpos is not None, a second additional attribute 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 the largest clique) for G. 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.

3.3.4 Clustering

triangles (G[, nbunch, with_labels])	Compute the number of triangles.
transitivity (G)	Compute transitivity.
clustering (G[, nbunch, with_labels, ...])	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.

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

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.

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.

3.3.5 Cores

`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.

3.3.6 Isomorphism

Approximate Isomorphism

graph_could_be_isomorphic (G1, G2)	Returns False if graphs G1 and G2 are definitely not isomorphic.
fast_graph_could_be_isomorphic (G1, G2)	Returns False if graphs G1 and G2 are definitely not isomorphic.
faster_graph_could_be_isomorphic (G1, G2)	Returns False if graphs G1 and G2 are definitely not isomorphic.
is_isomorphic (G1, G2)	Returns True if the graphs G1 and G2 are isomorphic and False otherwise.

networkx.graph_could_be_isomorphic

graph_could_be_isomorphic (*G1, G2*)

Returns False if graphs G1 and G2 are definitely not isomorphic.

True does NOT garantee isomorphism.

Checks for matching degree, triangle, and number of cliques sequences.

networkx.fast_graph_could_be_isomorphic

fast_graph_could_be_isomorphic (*G1, G2*)

Returns False if graphs G1 and G2 are definitely not isomorphic.

True does NOT garantee isomorphism.

Checks for matching degree and triangle sequences.

networkx.faster_graph_could_be_isomorphic

faster_graph_could_be_isomorphic (*G1, G2*)

Returns False if graphs G1 and G2 are definitely not isomorphic.

True does NOT garantee isomorphism.

Checks for matching degree sequences in G1 and G2.

networkx.is_isomorphic

is_isomorphic (*G1, G2*)

Returns True if the graphs G1 and G2 are isomorphic and False otherwise.

Uses the vf2 algorithm - see networkx.isomorphvf2

VF2 Algorithm

GraphMatcher	Check isomorphism of graphs.
DiGraphMatcher	Check isomorphism of directed graphs.

networkx.GraphMatcher

class **GraphMatcher** (*G1, G2*)

Check isomorphism of graphs.

A GraphMatcher is responsible for matching undirected graphs (Graph or XGraph) in a predetermined manner. For graphs G1 and G2, this typically means a check for an isomorphism between them, though other checks are also possible. For example, the GraphMatcher class can check if a subgraph of G1 is isomorphic to G2.

Matching is done via syntactic feasibility. It is also possible to check for semantic feasibility. Feasibility, then, is defined as the logical AND of the two functions.

To include a semantic check, the GraphMatcher class should be subclassed, and the semantic_feasibility() function should be redefined. By default, the semantic feasibility function always returns True. The effect of this is that semantics are not considered in the matching of G1 and G2.

For more information, see the docmentation for: syntactic_feasibliity() semantic_feasibility()

Notes

Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, Mario Vento, "A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 26, no. 10, pp. 1367-1372, Oct., 2004.

Modified to handle undirected graphs. Modified to handle multiple edges.

Examples

Suppose G1 and G2 are isomorphic graphs. Verification is as follows:

```
>>> G1=nx.path_graph(4)
>>> G2=nx.path_graph(4)
>>> GM = nx.GraphMatcher(G1,G2)
>>> GM.is_isomorphic()
True
>>> GM.mapping
{0: 0, 1: 1, 2: 2, 3: 3}
```

GM.mapping stores the isomorphism mapping.

networkx.DiGraphMatcher

class **DiGraphMatcher** (*G1, G2*)

Check isomorphism of directed graphs.

A DiGraphMatcher is responsible for matching directed graphs (DiGraph or XDiGraph) in a predetermined manner. For graphs G1 and G2, this typically means a check for an isomorphism between them, though other checks are also possible. For example, the DiGraphMatcher class can check if a subgraph of G1 is isomorphic to G2.

Matching is done via syntactic feasibility. It is also possible to check for semantic feasibility. Feasibility, then, is defined as the logical AND of the two functions.

To include a semantic check, you should subclass the GraphMatcher class and redefine semantic_feasibility(). By default, the semantic feasibility function always returns True. The effect of this is that semantics are not considered in the matching of G1 and G2.

For more information, see the docmentation for: syntactic_feasibliity() semantic_feasibility()

Suppose G1 and G2 are isomorphic graphs. Verfication is as follows:

```
>>> G1=nx.path_graph(4)
>>> G2=nx.path_graph(4)
>>> GM = nx.GraphMatcher(G1,G2)
>>> GM.is_isomorphic()
True
>>> GM.mapping
{0: 0, 1: 1, 2: 2, 3: 3}
```

GM.mapping stores the isomorphism mapping.

3.3.7 Traversal

Components

Undirected Graphs

is_connected (G)	Return True if G is connected. For undirected graphs only.
number_connected_components (G)	Return the number of connected components in G. For undirected graphs only.
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

```
>>> 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 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[, source])	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 connected components in G.

For undirected 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

`topological_sort` (G)	Return a list of nodes of the digraph G in topological sort order.
`topological_sort_recursive` (G)	Return a list of nodes of the digraph G 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*)

Return a list of nodes of the digraph G 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.

If G is not a directed acyclic graph 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/

See also is_directed_acyclic_graph()

networkx.topological_sort_recursive

`topological_sort_recursive`(*G*)

Return a list of nodes of the digraph G in topological sort order.

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).

Otherwise return False.

Distance

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.

Paths

`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 the source and target.
`bidirectional_shortest_path` (G, source, target)	Return list of nodes in a shortest path between 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[, cutoff])	Return the shortest path length from source to all reachable nodes.
`all_pairs_shortest_path` (G[, cutoff])	Return dictionary of 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 graph G.
`dijkstra_path_length` (G, source, target)	Returns the shortest path length from source to target in a weighted graph G.
`bidirectional_dijkstra` (G, source, target)	Dijkstra's algorithm for shortest paths using bidirectional search.
`single_source_dijkstra_path` (G, source)	Returns the shortest paths from source to all other reachable nodes in a weighted graph G.
`single_source_dijkstra_path_length` (G, source)	Returns the shortest path lengths from source to all other reachable nodes in a weighted graph G.
`single_source_dijkstra` (G, source[, target])	Dijkstra's algorithm for shortest paths in a weighted graph G.
`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 nodes in G.
`floyd_warshall` (G[, huge])	The Floyd-Warshall algorithm for all pairs shortest paths.

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.

networkx.shortest_path_length

shortest_path_length(*G, source, target*)

Return the shortest path length the source and target.

Raise an exception if no path exists.

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

networkx.bidirectional_shortest_path

bidirectional_shortest_path(*G, source, target*)

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

Return False if no path exists.

Also known as shortest_path.

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.

cutoff is optional integer depth to stop the search - only paths of length <= cutoff are returned.

See also shortest_path and bidirectional_shortest_path.

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.

```
>>> G=nx.path_graph(5)
>>> length=nx.single_source_shortest_path_length(G,1)
>>> length[4]
3
>>> print length
{0: 1, 1: 0, 2: 1, 3: 2, 4: 3}
```

cutoff is optional integer depth to stop the search - only paths of length <= cutoff are returned.

networkx.all_pairs_shortest_path

all_pairs_shortest_path(*G, cutoff=None*)

Return dictionary of shortest paths between all nodes.

The dictionary only has keys for reachable node pairs.

cutoff is optional integer depth to stop the search - only paths of length <= cutoff are returned.

See also floyd_warshall.

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. >>> 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}

cutoff is optional integer depth to stop the search - only paths of length <= cutoff are returned.

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.

Edge data must be numerical values for XGraph and XDiGraphs. The weights are assigned to be 1 for Graphs and DiGraphs.

See also bidirectional_dijkstra for more information about the algorithm.

networkx.dijkstra_path_length

dijkstra_path_length (*G, source, target*)

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

Uses a bidirectional version of Dijkstra's algorithm.

Edge data must be numerical values for XGraph and XDiGraphs. The weights are assigned to be 1 for Graphs and DiGraphs.

See also bidirectional_dijkstra for more information about the algorithm.

networkx.bidirectional_dijkstra

bidirectional_dijkstra (*G, source, target*)

Dijkstra's algorithm for shortest paths using bidirectional search.

Returns a two-tuple (d,p) where d is the distance and p is the path from the source to the target.

Distances are calculated as sums of weighted edges traversed.

Edges must hold numerical values for XGraph and XDiGraphs. The weights are set to 1 for Graphs 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).

networkx.single_source_dijkstra_path

single_source_dijkstra_path (*G, source*)

Returns the shortest paths from source to all other reachable nodes in a weighted graph G.

Uses Dijkstra's algorithm.

Returns a dictionary of shortest path lengths keyed by source.

Edge data must be numerical values for XGraph and XDiGraphs. The weights are assigned to be 1 for Graphs and DiGraphs.

See also single_source_dijkstra for more information about the algorithm.

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.

Uses Dijkstra's algorithm.

Returns a dictionary of shortest path lengths keyed by source.

Edge data must be numerical values for XGraph and XDiGraphs. The weights are assigned to be 1 for Graphs and DiGraphs.

See also single_source_dijkstra for more information about the algorithm.

networkx.single_source_dijkstra

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

Dijkstra's algorithm for shortest paths in a weighted graph G.

Use:

single_source_dijkstra_path() - shortest path list of nodes

single_source_dijkstra_path_length() - shortest path length

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

Distances are calculated as sums of weighted edges traversed. Edges must hold numerical values for XGraph and XDiGraphs. The weights are 1 for Graphs and DiGraphs.

Optional target argument stops the search when target is found.

Based on the Python cookbook recipe (119466) at http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/119

This algorithm is not guaranteed to work if edge weights are negative or are floating point numbers (overflows and roundoff errors can cause problems).

See also 'bidirectional_dijkstra_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.

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.

Optional target returns only predecessors between source and target. Cutoff is a limit on the number of hops traversed.

Example for the path graph 0-1-2-3

```
>>> 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.

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.

For sparse graphs, see

all_pairs_shortest_path all_pairs_shortest_path_length

which are based on Dijkstra's algorithm.

Search

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 Graph generators

3.4.1 Atlas

graph_atlas_g ()	Return the list [G0,G1,...,G1252] of graphs as named in the Graph Atlas. G0,G1,...,G1252 are all graphs with up to 7 nodes.

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)

3.4.2 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. Optional argument periodic=True will connect boundary nodes via periodic boundary conditions.
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 by n-1 edges.
star_graph (n)	Return the Star graph with n+1 nodes: one center node, connected to n outer nodes.
trivial_graph ()	Return the Trivial graph with one node (with integer label 0) and no edges.
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**
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 whee**

to each node of the (n-1)-node cycle graph.

Node labels are the integers 0 to n - 1.

3.4.3 Small

make_small_graph (graph_description[, create_using])	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.

3.4.4 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.
random_regular_graph (d, n[, seed])	Return a random regular graph of n nodes each with degree d, G_{n,d}. Return False if unsuccessful.
barabasi_albert_graph (n, m[, seed])	Return random graph using Barabási-Albert preferential attachment model.
powerlaw_cluster_graph (n, m, p[, seed])	Holme and Kim algorithm for growing graphs with powerlaw degree distribution and approximate average clustering.
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, seed, tries])	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 rewiring existing edges as follows: for each edge u-v in the underlying "n-ring with k nearest neighbors" with probability p replace u-v with a new edge u-w with randomly-chosen existing node w. In contrast with newman_watts_strogatz_graph(), the random rewiring does not increase the number of edges.

Parameters

- *n*: the number of nodes
- *k*: each node is connected to k neighbors in the ring topology
- *p*: the probability of rewiring an edge
- *seed*: seed for random number generator (default=None)

networkx.random_regular_graph

random_regular_graph (*d, n, seed=None*)

Return a random regular graph of n nodes each with degree d, G_{n,d}. Return False if unsuccessful.

n*d must be even

Nodes are numbered 0...n-1. To get a uniform sample from the space of random graphs you should chose d<n^{1/3}.

For algorith see Kim and Vu's paper.

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},
}
```

The algorithm is based on an earlier paper:

```
@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",
}
```

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*: the number of nodes
- *m*: number of edges to attach from a new node to existing nodes
- *seed*: seed for random number generator (default=None)

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

Reference:

```
@article{barabasi-1999-emergence,
title   = {Emergence of scaling in random networks},
author  = {A. L. Barabási and R. Albert},
journal = {Science},
volume  = {286},
number  = {5439},
pages   = {509 -- 512},
year = {1999},
}
```

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)

3.4.5 Degree Sequence

`configuration_model` (deg_sequence[, seed])	Return a random pseudograph 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 Havel-Hakimi algorithm.
`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 equal to the degree sequence of some simple graph.
`create_degree_sequence` (n[, sfunction, max_tries, **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. Experimental implementation.
`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

networkx.configuration_model

`configuration_model`(*deg_sequence, seed=None*)

Return a random pseudograph with the given degree sequence.

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

```
>>> from networkx.utils import powerlaw_sequence
>>> z=nx.create_degree_sequence(100,powerlaw_sequence)
>>> G=nx.configuration_model(z)
```

The pseudograph G is a networkx.MultiGraph that allows multiple (parallel) edges between nodes and self-loops (edges from a node to itself).

To remove parallel edges:

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

Steps:

- Check if deg_sequence is a valid degree sequence.
- Create N nodes with stubs for attaching edges
- Randomly select two available stubs and connect them with an edge.

As described by Newman [newman-2003-structure].

Nodes are labeled 1,.., len(deg_sequence), corresponding to their position in deg_sequence.

This process can lead to duplicate edges and loops, and therefore returns a pseudograph type. You can remove the self-loops and parallel edges (see above) with the likely result of not getting the exat degree sequence specified. This "finite-size effect" decreases as the size of the graph increases.

References:

[newman-2003-structure] M.E.J. Newman, "The structure and function of complex networks", SIAM REVIEW 45-2, pp 167-256, 2003.

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}
 }
```

3.4.6 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 and redirection probability p.
gnc_graph (n[, seed])	Return the GNC (growing network with copying) digraph with n nodes.

networkx.gn_graph

gn_graph (*n, kernel=<function <lambda> at 0x9061a74>, 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},
}
```

3.4.7 Geometric

random_geometric_graph (n, radius[, create_using, repel, ...])	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.

3.4.8 Hybrid

`kl_connected_subgraph` (G, k, l[, low_memory, same_as_graph])	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 Linear Algebra

3.5.1 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

3.6 Reading and Writing

3.6.1 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, delimiter, ...])	Read graph in multi-line adjacency list format from path.
`write_multiline_adjlist` (G, path[, delimiter, comments])	Write the graph G in multiline adjacency list format to the file or file handle path.

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='utf=8') # 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", edgetype=int)
```

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.

```
>>> 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

```
>>> 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.

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

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

```
>>> 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.

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

3.6.2 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

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")
```

3.6.3 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. Returns an Graph or DiGraph.
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. Returns an Graph or DiGraph.

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.

3.6.4 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.

3.6.5 GraphML

Read graphs in GraphML format. http://graphml.graphdrawing.org/

read_graphml (path)	Read graph in GraphML format from path. Returns an Graph or DiGraph.
parse_graphml (lines)	Read graph in GraphML format from string. Returns an Graph or DiGraph.

networkx.read_graphml

read_graphml (*path*)

Read graph in GraphML format from path. Returns an Graph or DiGraph.

networkx.parse_graphml

parse_graphml (*lines*)

Read graph in GraphML format from string. Returns an Graph or DiGraph.

3.6.6 LEDA

read_leda (path)	Read graph in GraphML format from path. Returns an XGraph or XDiGraph.
parse_leda (lines)	Parse LEDA.GRAPH format from string or iterable. Returns an Graph or DiGraph.

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.

3.6.7 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[, default_flow_style, **kwds)	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

3.6.8 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. Returns a single Graph.
parse_graph6 (str)	Read undirected graph in graph6 format.
read_graph6_list (path)	Read simple undirected graphs in graph6 format from path. Returns a list of Graphs, one for each line in file.
read_sparse6 (path)	Read simple undirected graphs in sparse6 format from path. Returns a single Graph.
parse_sparse6 (str)	Read undirected graph in sparse6 format.
read_sparse6_list (path)	Read simple undirected graphs in sparse6 format from path. Returns a list of Graphs, one for each line in file.

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.

3.7 Drawing

3.7.1 Matplotlib

Draw networks with matplotlib (pylab).

References:

- matplotlib: http://matplotlib.sourceforge.net/
- pygraphviz: http://networkx.lanl.gov/pygraphviz/

draw (G[, pos, ax, hold, **kwds)	Draw the graph G with matplotlib (pylab).
draw_networkx (G, pos[, with_labels, **kwds)	Draw the graph G with given node positions pos
draw_networkx_nodes (G, pos[, nodelist, node_size, node_color, node_shape, alpha, cmap, vmin, vmax, ax, linewidths, **kwds)	Draw nodes of graph G
draw_networkx_edges (G, pos[, edgelist, width, edge_color, style, alpha, edge_cmap, edge_vmin, edge_vmax, ax, arrows, **kwds)	Draw the edges of the graph G
draw_networkx_labels (G, pos[, labels, font_size, font_color, font_family, font_weight, alpha, ax, **kwds)	Draw node labels on the graph G
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[, prog, **kwargs)	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=1.0, 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.

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_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

3.7.2 Graphviz AGraph (dot)

Interface to pygraphviz AGraph class.

Usage

```
>>> G=nx.complete_graph(5)
>>> A=nx.to_agraph(G)
>>> H=nx.from_agraph(A)
```

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

`from_agraph` (A[, create_using])	Return a NetworkX Graph or DiGraph from a pygraphviz graph.
`to_agraph` (N[, graph_attr, node_attr, ...])	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 XGraph or XdiGraph from a dot file on path.
`graphviz_layout` (G[, prog, root, args])	Create layout using graphviz. Returns a dictionary of positions keyed by node.
`pygraphviz_layout` (G[, prog, root, args])	Create layout using pygraphviz and graphviz. Returns a dictionary of positions keyed by node.

networkx.from_agraph

from_agraph (*A, create_using=None*)

Return a NetworkX Graph or DiGraph from a pygraphviz graph.

```
>>> G=nx.complete_graph(5)
>>> A=nx.to_agraph(G)
>>> X=nx.from_agraph(A)
```

The Graph X will have a dictionary X.graph_attr containing the default graphviz attributes for graphs, nodes and edges.

Default node attributes will be in the dictionary X.node_attr which is keyed by node.

Edge attributes will be returned as edge data in the graph X.

If you want a Graph with no attributes attached instead of an XGraph with attributes use

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

networkx.to_agraph

to_agraph (*N, graph_attr=None, node_attr=None, strict=True*)

Return a pygraphviz graph from a NetworkX graph N.

If N is a Graph or DiGraph, graphviz attributes can be supplied through the arguments

graph_attr: dictionary with default attributes for graph, nodes, and edges keyed by 'graph', 'node', and 'edge' to attribute dictionaries

node_attr: dictionary keyed by node to node attribute dictionary

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.

networkx.write_dot

write_dot (*G, path*)

Write NetworkX graph G to Graphviz dot format on path.

Path can be a string or a file handle.

networkx.read_dot

read_dot (*path, create_using=None*)

Return a NetworkX XGraph or XdiGraph from a dot file on path.

Path can be a string or a file handle.

networkx.graphviz_layout

graphviz_layout (*G, prog='neato', root=None, args=''*)

Create layout using graphviz. Returns a dictionary of positions keyed by node.

```
>>> G=nx.petersen_graph()
>>> pos=nx.graphviz_layout(G)
>>> pos=nx.graphviz_layout(G,prog='dot')
```

This is a wrapper for pygraphviz_layout.

networkx.pygraphviz_layout

pygraphviz_layout (*G, prog='neato', root=None, args=''*)

Create layout using pygraphviz and graphviz. Returns a dictionary of positions keyed by node.

```
>>> G=nx.petersen_graph()
>>> pos=nx.pygraphviz_layout(G)
>>> pos=nx.pygraphviz_layout(G,prog='dot')
```

3.7.3 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.

References:

- pydot Homepage: 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 or DiGraph from a pydot graph.
to_pydot (N[, graph_attr, node_attr, ...])	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 XGraph or XdiGraph from a dot file on path.
graphviz_layout (G[, prog, root, args])	Create layout using graphviz. Returns a dictionary of positions keyed by node.
pydot_layout (G[, prog, root, **kwds)	Create layout using pydot and graphviz. Returns a dictionary of positions keyed by node.

networkx.from_pydot

from_pydot (*P*)

Return a NetworkX Graph or DiGraph from a pydot graph.

The Graph X will have a dictionary X.graph_attr containing the default graphviz attributes for graphs, nodes and edges.

Default node attributes will be in the dictionary X.node_attr which is keyed by node.

Edge attributes will be returned as edge data in the graph X.

Examples

```
>>> G=nx.complete_graph(5)
>>> P=nx.to_pydot(G)
>>> X=nx.from_pydot(P)
```

If you want a Graph with no attributes attached use

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

Similarly to make a DiGraph without attributes

```
>>> D=nx.DiGraph(X)
```

networkx.to_pydot

to_pydot (*N, graph_attr=None, node_attr=None, edge_attr=None, strict=True*)

Return a pydot graph from a NetworkX graph N.

If N is a Graph or DiGraph, graphviz attributes can be supplied through the keyword arguments

graph_attr: dictionary with default attributes for graph, nodes, and edges keyed by 'graph', 'node', and 'edge' to attribute dictionaries node_attr: dictionary keyed by node to node attribute dictionary

edge_attr: dictionary keyed by edge tuple to edge attribute dictionary

If N is an XGraph or XDiGraph an attempt will be made first to copy properties attached to the graph (see from_pydot) and then updated with the calling arguments, if any.

networkx.write_dot

write_dot (*G, path*)

Write NetworkX graph G to Graphviz dot format on path.

Path can be a string or a file handle.

networkx.read_dot

read_dot (*path, create_using=None*)

Return a NetworkX XGraph or XdiGraph from a dot file on path.

Path can be a string or a file handle.

networkx.graphviz_layout

graphviz_layout (*G, prog='neato', root=None, args=''*)

Create layout using graphviz. Returns a dictionary of positions keyed by node.

```
>>> G=nx.petersen_graph()
>>> pos=nx.graphviz_layout(G)
>>> pos=nx.graphviz_layout(G,prog='dot')
```

This is a wrapper for pygraphviz_layout.

networkx.pydot_layout

pydot_layout (*G, prog='neato', root=None, **kwds*)

Create layout using pydot and graphviz. Returns a dictionary of positions keyed by node.

```
>>> G=nx.complete_graph(4)
>>> pos=nx.pydot_layout(G)
>>> pos=nx.pydot_layout(G,prog='dot')
```

3.7.4 Graph Layout

Node positioning algorithms for graph drawing.

`circular_layout` (G[, dim])	Circular layout.
`random_layout` (G[, dim])	Random layout.
`shell_layout` (G[, nlist, dim])	Shell layout. Crude version that doesn't try to minimize edge crossings.
`spring_layout` (G[, iterations, dim, ...])	Spring force model layout
`spectral_layout` (G[, dim, vpos, iterations, ...])	Return the position vectors for drawing G using spectral layout.

networkx.circular_layout

circular_layout (*G, dim=2*)
Circular layout.

Crude version that doesn't try to minimize edge crossings.

networkx.random_layout

random_layout (*G, dim=2*)
Random layout.

networkx.shell_layout

shell_layout (*G, nlist=None, dim=2*)
Shell layout. Crude version that doesn't try to minimize edge crossings.

nlist is an optional list of lists of nodes to be drawn at each shell level. Only one shell with all nodes will be drawn if not specified.

networkx.spring_layout

spring_layout (*G, iterations=50, dim=2, node_pos=None*)
Spring force model layout

networkx.spectral_layout

spectral_layout (*G, dim=2, vpos=None, iterations=1000, eps=0.001*)
Return the position vectors for drawing G using spectral layout.

3.8 History

NetworkX = Network "X" = NX (for short)

Original Creators:

```
Aric Hagberg, hagberg@lanl.gov
Pieter Swart, swart@lanl.gov
Dan Schult, dschult@colgate.edu
```


3.8.1 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_edges()

Now raises an exception on an attempt to delete an edge not in the graph. The preferred name is now remove_edges().

delete_edges_from()

The preferred name is now remove_edge().

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:

```
import networkx.centrality
from networkx.centrality import *
```

may no longer work (including that example).

Use either

```
>>> import networkx # e.g. centrality functions available as networkx.fcn()
```

or

```
>>> 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.

3.8.2 Release Log

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/

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

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

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.

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

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_*

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

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_si
  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

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

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

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

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

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()

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)

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

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

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.

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

3.9 Credits

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 the following people who have made contributions to NetworkX:

- 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
- Eben Kennah contributed the strongly connected components and DFS functions.
- Sasha Gutfriend contributed edge betweenness algorithms

3.10 Legal

3.10.1 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
```

3.10.2 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.
```

3.11 Citing

To cite NetworkX please use the following publication:

Aric A. Hagberg, Daniel A. Schult and Pieter J. Swart, "Exploring network structure, dynamics, and func-

tion using NetworkX", in Proceedings of the 7th Python in Science Conference (SciPy2008), Gäel Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11–15, Aug 2008

CHAPTER

FOUR

DOWNLOAD

4.1 Source and Binary Releases

http://cheeseshop.python.org/pypi/networkx/

http://networkx.lanl.gov/download/networkx/

4.2 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

4.3 Documentation

PDF

http://networkx.lanl.gov/networkx/networkx.pdf

HTML in zip file

http://networkx.lanl.gov/networkx/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.

MODULE INDEX

N

INDEX

Symbols

A

F

G

H

I

K

L

M

N

O

P

R

S

T

W