
NetworkX Reference

Release 1.2

Aric Hagberg, Dan Schult, Pieter Swart

August 01, 2010

CONTENTS

1	Introduction	1
1.1	Who uses NetworkX?	1
1.2	The Python programming language	1
1.3	Free software	1
1.4	Goals	1
1.5	History	2
2	Overview	3
2.1	NetworkX Basics	3
2.2	Nodes and Edges	4
3	Graph types	9
3.1	Which graph class should I use?	9
3.2	Basic graph types	9
4	Algorithms	133
4.1	Bipartite	133
4.2	Blockmodeling	135
4.3	Boundary	136
4.4	Centrality	137
4.5	Clique	145
4.6	Clustering	147
4.7	Components	150
4.8	Cores	158
4.9	Cycles	158
4.10	Directed Acyclic Graphs	159
4.11	Distance Measures	161
4.12	Eulerian	162
4.13	Flows	164
4.14	Isolates	166
4.15	Isomorphism	168
4.16	Link Analysis	179
4.17	Matching	185
4.18	Mixing Patterns	186
4.19	Minimum Spanning Tree	191
4.20	Operators	191
4.21	Shortest Paths	195
4.22	Traversal	209
4.23	Vitality	210

5	Functions	213
5.1	Graph functions	213
6	Graph generators	217
6.1	Atlas	217
6.2	Classic	217
6.3	Small	221
6.4	Random Graphs	226
6.5	Degree Sequence	236
6.6	Directed	243
6.7	Geometric	247
6.8	Hybrid	247
6.9	Bipartite	247
6.10	Line Graph	251
6.11	Ego Graph	252
6.12	Stochastic	252
7	Linear algebra	253
7.1	Spectrum	253
7.2	Attribute Matrices	255
8	Converting to and from other data formats	259
8.1	To NetworkX Graph	259
8.2	Relabeling	260
8.3	Dictionaries	261
8.4	Lists	262
8.5	Numpy	264
8.6	Scipy	265
9	Reading and writing graphs	269
9.1	Adjacency List	269
9.2	Edge List	273
9.3	GML	277
9.4	Pickle	279
9.5	GraphML	280
9.6	LEDA	281
9.7	YAML	281
9.8	SparseGraph6	281
9.9	Pajek	282
10	Drawing	285
10.1	Matplotlib	285
10.2	Graphviz AGraph (dot)	293
10.3	Graphviz with pydot	296
10.4	Graph Layout	299
11	Exceptions	303
12	Utilities	305
12.1	Helper functions	305
12.2	Data structures and Algorithms	306
12.3	Random sequence generators	306
12.4	SciPy random sequence generators	307
13	License	309

14 Citing	311
15 Credits	313
16 Glossary	315
Bibliography	317
Module Index	321
Index	323

INTRODUCTION

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

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

1.1 Who uses NetworkX?

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

1.2 The Python programming language

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

1.3 Free software

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

1.4 Goals

NetworkX is intended to:

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

1.5 History

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

1.5.1 What Next

- A Brief Tour
- Installing
- Reference
- Examples

OVERVIEW

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

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

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

2.1 NetworkX Basics

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

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

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

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

The following basic graph types are provided as Python classes:

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

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

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

MultiDiGraph A directed version of a MultiGraph.

Empty graph-like objects are created with

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

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

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

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

2.1.1 Graphs

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

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

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

2.2 Nodes and Edges

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

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

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

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

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

2.2.1 Graph Creation

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

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

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

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

Edge attributes can be anything:

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

You can add many edges at one time:

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

See the [/tutorial/index](#) for more examples.

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

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

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

2.2.2 Graph Reporting

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

The basic graph relationship of an edge can be obtained in two basic ways. One can look for neighbors of a node or one can look for edges incident to a node. We jokingly refer to people who focus on nodes/neighbors as node-centric and people who focus on edges as edge-centric. The designers of NetworkX tend to be node-centric and view edges as a relationship between nodes. You can see this by our avoidance of notation like $G[u,v]$ in favor of $G[u][v]$. Most data structures for sparse graphs are essentially adjacency lists and so fit this perspective. In the end, of course, it doesn't really matter which way you examine the graph. `G.edges()` removes duplicate representations of each edge while `G.neighbors(n)` or `G[n]` is slightly faster but doesn't remove duplicates.

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

2.2.3 Algorithms

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

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

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

2.2.4 Drawing

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

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

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

See the `examples` for more ideas.

2.2.5 Data Structure

NetworkX uses a “dictionary of dictionaries of dictionaries” as the basic network data structure. This allows fast lookup with reasonable storage for large sparse networks. The keys are nodes so `G[u]` returns an adjacency dictionary keyed by neighbor to the edge attribute dictionary. The expression `G[u][v]` returns the edge attribute dictionary itself. A dictionary of lists would have also been possible, but not allowed fast edge detection nor convenient storage of edge data.

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

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

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

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

The data structure gets morphed slightly for each base graph class. For DiGraph two dict-of-dicts-of-dicts structures are provided, one for successors and one for predecessors. For MultiGraph/MultiDiGraph we use a dict-of-dicts-of-dicts¹ where the third dictionary is keyed by an edge key identifier to the fourth dictionary which contains the edge attributes for that edge between the two nodes.

Graphs use a dictionary of attributes for each edge. We use a dict-of-dicts-of-dicts data structure with the inner dictionary storing “name-value” relationships for that edge.

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

¹ “It’s dictionaries all the way down.”

GRAPH TYPES

NetworkX provides data structures and methods for storing graphs.

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

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

3.1 Which graph class should I use?

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

3.2 Basic graph types

3.2.1 Graph – Undirected graphs with self loops

Overview

Graph (*data=None, name="", **attr*)

Base class for undirected graphs.

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

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

Nodes can be arbitrary (hashable) Python objects with optional key/value attributes.

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

Parameters **data** : input graph

Data to initialize graph. If data=None (default) an empty graph is created. The data can be an edge list, or any NetworkX graph object. If the corresponding optional Python packages are installed the data can also be a NumPy matrix or 2d ndarray, a SciPy sparse matrix, or a PyGraphviz graph.

name : string, optional (default='')

An optional name for the graph.

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

Attributes to add to graph as key=value pairs.

See Also:

`DiGraph`, `MultiGraph`, `MultiDiGraph`

Examples

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

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

G can be grown in several ways.

Nodes:

Add one node at a time:

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

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

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

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

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

Edges:

G can also be grown by adding edges.

Add one edge,

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

a list of edges,

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

or a collection of edges,

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

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

Attributes:

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

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

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

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

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

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

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

Shortcuts:

Many common graph features allow python syntax to speed reporting.

```
>>> 1 in G      # check if node in graph
True
>>> print [n for n in G if n<3]  # iterate through nodes
[1, 2]
>>> print len(G)  # number of nodes in graph
5
>>> print G[1] # adjacency dict keyed by neighbor to edge attributes
...          # Note: you should not change this dict manually!
{2: {'color': 'blue', 'weight': 4}}
```

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

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

Reporting:

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

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

Adding and removing nodes and edges

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

networkx.Graph.__init__

`__init__(data=None, name="", **attr)`
 Initialize a graph with edges, name, graph attributes.

Parameters **data** : input graph

Data to initialize graph. If data=None (default) an empty graph is created. The data can be an edge list, or any NetworkX graph object. If the corresponding optional Python packages are installed the data can also be a NumPy matrix or 2d ndarray, a SciPy sparse matrix, or a PyGraphviz graph.

name : string, optional (default='')

An optional name for the graph.

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

Attributes to add to graph as key=value pairs.

See Also:

convert

Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G = nx.Graph(name='my graph')
>>> e = [(1,2), (2,3), (3,4)] # list of edges
>>> G = nx.Graph(e)
```

Arbitrary graph attribute pairs (key=value) may be assigned

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

networkx.Graph.add_node

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

Add a single node *n* and update node attributes.

Parameters *n* : node

A node can be any hashable Python object except None.

attr_dict : dictionary, optional (default= no attributes)

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

attr : keyword arguments, optional

Set or change attributes using key=value.

See Also:

[add_nodes_from](#)

Notes

A hashable object is one that can be used as a key in a Python dictionary. This includes strings, numbers, tuples of strings and numbers, etc.

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

Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_node(1)
>>> G.add_node('Hello')
>>> K3 = nx.Graph([(0,1), (1,2), (2,0)])
>>> G.add_node(K3)
>>> G.number_of_nodes()
3
```

Use keywords set/change node attributes:

```
>>> G.add_node(1, size=10)
>>> G.add_node(3, weight=0.4, UTM=('13S', 382871, 3972649))
```

networkx.Graph.add_nodes_from

add_nodes_from(nodes, ***attr*)

Add multiple nodes.

Parameters nodes : iterable container

A container of nodes (list, dict, set, etc.). OR A container of (node, attribute dict) tuples.
Node attributes are updated using the attribute dict.

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

Update attributes for all nodes in nodes. Node attributes specified in nodes as a tuple take precedence over attributes specified generally.

See Also:

`add_node`

Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_nodes_from('Hello')
>>> K3 = nx.Graph([(0,1), (1,2), (2,0)])
>>> G.add_nodes_from(K3)
>>> sorted(G.nodes())
[0, 1, 2, 'H', 'e', 'l', 'o']
```

Use keywords to update specific node attributes for every node.

```
>>> G.add_nodes_from([1,2], size=10)
>>> G.add_nodes_from([3,4], weight=0.4)
```

Use (node, attrdict) tuples to update attributes for specific nodes.

```
>>> G.add_nodes_from([(1,dict(size=11)), (2,{'color':'blue'})])
>>> G.node[1]['size']
11
>>> H = nx.Graph()
>>> H.add_nodes_from(G.nodes(data=True))
>>> H.node[1]['size']
11
```

networkx.Graph.remove_node

remove_node(n)

Remove node n.

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

Parameters n : node

A node in the graph

Raises NetworkXError :

If n is not in the graph.

See Also:`remove_nodes_from`**Examples**

```

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

```

networkx.Graph.remove_nodes_from**remove_nodes_from**(nodes)

Remove multiple nodes.

Parameters nodes : iterable container

A container of nodes (list, dict, set, etc.). If a node in the container is not in the graph it is silently ignored.

See Also:`remove_node`**Examples**

```

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

```

networkx.Graph.add_edge**add_edge**(u, v, attr_dict=None, **attr)

Add an edge between u and v.

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

Edge attributes can be specified with keywords or by providing a dictionary with key/value pairs. See examples below.

Parameters u,v : nodes

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

attr_dict : dictionary, optional (default= no attributes)

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

attr : keyword arguments, optional

Edge data (or labels or objects) can be assigned using keyword arguments.

See Also:

[add_edges_from](#) add a collection of edges

Notes

Adding an edge that already exists updates the edge data.

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

Examples

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

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> e = (1,2)
>>> G.add_edge(1, 2) # explicit two-node form
>>> G.add_edge(*e) # single edge as tuple of two nodes
>>> G.add_edges_from( [(1,2)] ) # add edges from iterable container
```

Associate data to edges using keywords:

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

networkx.Graph.add_edges_from

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

Add all the edges in ebunch.

Parameters **ebunch** : container of edges

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

attr_dict : dictionary, optional (default= no attributes)

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

attr : keyword arguments, optional

Edge data (or labels or objects) can be assigned using keyword arguments.

See Also:

[add_edge](#) add a single edge

`add_weighted_edges_from` convenient way to add weighted edges

Notes

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

Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edges_from([(0,1), (1,2)]) # using a list of edge tuples
>>> e = zip(range(0,3), range(1,4))
>>> G.add_edges_from(e) # Add the path graph 0-1-2-3
```

Associate data to edges

```
>>> G.add_edges_from([(1,2), (2,3)], weight=3)
>>> G.add_edges_from([(3,4), (1,4)], label='WN2898')
```

networkx.Graph.add_weighted_edges_from

`add_weighted_edges_from`(*ebunch*, ***attr*)

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

Parameters *ebunch* : container of edges

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

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

Edge attributes to add/update for all edges.

See Also:

`add_edge` add a single edge

`add_edges_from` add multiple edges

Notes

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

Examples

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

networkx.Graph.remove_edge

remove_edge (*u, v*)

Remove the edge between *u* and *v*.

Parameters *u,v: nodes* :

Remove the edge between nodes *u* and *v*.

Raises **NetworkXError** :

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

See Also:

[remove_edges_from](#) remove a collection of edges

Examples

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

networkx.Graph.remove_edges_from

remove_edges_from (*ebunch*)

Remove all edges specified in *ebunch*.

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

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

- 2-tuples (*u,v*) edge between *u* and *v*.
- 3-tuples (*u,v,k*) where *k* is ignored.

See Also:

[remove_edge](#) remove a single edge

Notes

Will fail silently if an edge in *ebunch* is not in the graph.

Examples


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

networkx.Graph.add_star

add_star (*nlist*, ****attr**)

Add a star.

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

Parameters *nlist* : list

A list of nodes.

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

Attributes to add to every edge in star.

See Also:

`add_path`, `add_cycle`

Examples

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

networkx.Graph.add_path

add_path (*nlist*, ****attr**)

Add a path.

Parameters *nlist* : list

A list of nodes. A path will be constructed from the nodes (in order) and added to the graph.

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

Attributes to add to every edge in path.

See Also:

`add_star`, `add_cycle`

Examples

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

networkx.Graph.add_cycle

add_cycle (*nlist*, ***attr*)

Add a cycle.

Parameters *nlist* : list

A list of nodes. A cycle will be constructed from the nodes (in order) and added to the graph.

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

Attributes to add to every edge in cycle.

See Also:

`add_path`, `add_star`

Examples

```
>>> G=nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_cycle([0,1,2,3])
>>> G.add_cycle([10,11,12],weight=7)
```

networkx.Graph.clear

clear ()

Remove all nodes and edges from the graph.

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

Examples

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

Iterating over nodes and edges

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

networkx.Graph.nodes

nodes (*data=False*)

Return a list of the nodes in the graph.

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

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

Returns **nlist** : list

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

Examples

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

networkx.Graph.nodes_iter

nodes_iter (*data=False*)

Return an iterator over the nodes.

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

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

Returns **niter** : iterator

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

Notes

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

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

Examples

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

networkx.Graph.__iter__

`__iter__()`

Iterate over the nodes. Use the expression ‘for n in G’.

Returns niter : iterator

An iterator over all nodes in the graph.

Examples

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

networkx.Graph.edges

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

Return a list of edges.

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

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

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

data : bool, optional (default=False)

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

Returns edge_list: list of edge tuples :

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

See Also:

`edges_iter` return an iterator over the edges

Notes

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

Examples

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

networkx.Graph.edges_iter

edges_iter (nbunch=None, data=False)

Return an iterator over the edges.

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

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

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

data : bool, optional (default=False)

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

Returns edge_iter : iterator

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

See Also:

`edges` return a list of edges

Notes

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

Examples

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

networkx.Graph.get_edge_data

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

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

Parameters *u,v* : nodes

default: any Python object (default=None) :

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

Returns *edge_dict* : dictionary

The edge attribute dictionary.

Notes

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

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

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

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

Examples

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

```
>>> G.get_edge_data(*e) # tuple form
{}
>>> G.get_edge_data('a','b',default=0) # edge not in graph, return 0
0
```

networkx.Graph.neighbors

neighbors (*n*)

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

Parameters *n* : node

A node in the graph

Returns *nlist* : list

A list of nodes that are adjacent to *n*.

Raises **NetworkXError** :

If the node *n* is not in the graph.

Notes

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

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

Examples

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

networkx.Graph.neighbors_iter

neighbors_iter (*n*)

Return an iterator over all neighbors of node *n*.

Notes

It is faster to use the idiom “in *G*[0]”, e.g. `>>> for n in G[0]: ... print n`

Examples

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

networkx.Graph.__getitem__

__getitem__(*n*)

Return a dict of neighbors of node *n*. Use the expression 'G[n]'.

Parameters *n* : node

A node in the graph.

Returns *adj_dict* : dictionary

The adjacency dictionary for nodes connected to *n*.

Notes

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

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

Examples

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

networkx.Graph.adjacency_list

adjacency_list()

Return an adjacency list representation of the graph.

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

Returns *adj_list* : lists of lists

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

See Also:

`adjacency_iter`

Examples

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

networkx.Graph.adjacency_iter

adjacency_iter()

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

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

Returns `adj_iter` : iterator

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

See Also:

`adjacency_list`

Examples

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

networkx.Graph.nbunch_iter

nbunch_iter(*nbunch=None*)

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

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

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

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

Returns `niter` : iterator

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

Raises `NetworkXError` :

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

See Also:

`Graph.__iter__`

Notes

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

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

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

Information about graph structure

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

networkx.Graph.has_node

has_node (*n*)

Return True if the graph contains the node n.

Parameters *n* : node

Examples

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

It is more readable and simpler to use

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

networkx.Graph.__contains__

__contains__ (*n*)

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

Examples

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

networkx.Graph.has_edge

has_edge (*u, v*)

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

Parameters *u,v* : nodes

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

Returns *edge_ind* : bool

True if edge is in the graph, False otherwise.

Examples

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

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

The following syntax are all equivalent:

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

networkx.Graph.order

order ()

Return the number of nodes in the graph.

Returns *nnodes* : int

The number of nodes in the graph.

See Also:

`number_of_nodes`, `__len__`

`networkx.Graph.number_of_nodes`

`number_of_nodes()`

Return the number of nodes in the graph.

Returns `nnodes` : int

The number of nodes in the graph.

See Also:

`order`, `__len__`

Examples

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

`networkx.Graph.__len__`

`__len__()`

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

Returns `nnodes` : int

The number of nodes in the graph.

Examples

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

`networkx.Graph.degree`

`degree` (*`nbunch=None`, `weighted=False`*)

Return the degree of a node or nodes.

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

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

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

`weighted` : bool, optional (default=False)

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

Returns `nd` : dictionary, or number

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

Examples

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

networkx.Graph.degree_iter

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

Return an iterator for (node, degree).

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

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

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

weighted : bool, optional (default=False)

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

Returns `nd_iter` : an iterator

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

See Also:

`degree`

Examples

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

networkx.Graph.size

size (*weighted=False*)

Return the number of edges.

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

If True return the sum of the edge weights.

Returns `nedges` : int

The number of edges in the graph.

See Also:

`number_of_edges`

Examples

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

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge('a','b',weight=2)
>>> G.add_edge('b','c',weight=4)
>>> G.size()
2
>>> G.size(weighted=True)
6
```

`networkx.Graph.number_of_edges`

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

Return the number of edges between two nodes.

Parameters `u,v` : nodes, optional (default=all edges)

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

Returns `nedges` : int

The number of edges in the graph. If nodes `u` and `v` are specified return the number of edges between those nodes.

See Also:

`size`

Examples

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

```
>>> G.number_of_edges(*e)
1
```

networkx.Graph.nodes_with_selfloops

nodes_with_selfloops()

Return a list of nodes with self loops.

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

Returns nodelist : list

A list of nodes with self loops.

See Also:

`selfloop_edges`, `number_of_selfloops`

Examples

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

networkx.Graph.selfloop_edges

selfloop_edges(data=False)

Return a list of selfloop edges.

A selfloop edge has the same node at both ends.

Parameters data : bool, optional (default=False)

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

Returns edgelist : list of edge tuples

A list of all selfloop edges.

See Also:

`selfloop_nodes`, `number_of_selfloops`

Examples

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

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

networkx.Graph.number_of_selfloops

number_of_selfloops()

Return the number of selfloop edges.

A selfloop edge has the same node at both ends.

Returns nloops : int

The number of selfloops.

See Also:

`selfloop_nodes`, `selfloop_edges`

Examples

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

Making copies and subgraphs

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

networkx.Graph.copy

copy()

Return a copy of the graph.

Returns G : Graph

A copy of the graph.

See Also:

`to_directed` return a directed copy of the graph.

Notes

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

Examples

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

networkx.Graph.to_undirected

to_undirected()

Return an undirected copy of the graph.

Returns **G** : Graph/MultiGraph

A deepcopy of the graph.

See Also:

`copy`, `add_edge`, `add_edges_from`

Notes

This returns a “deepcopy” of the edge, node, and graph attributes which attempts to completely copy all of the data and references.

This is in contrast to the similar `G=DiGraph(D)` which returns a shallow copy of the data.

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

Examples

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

networkx.Graph.to_directed

to_directed()

Return a directed representation of the graph.

Returns **G** : DiGraph

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

Notes

This returns a “deepcopy” of the edge, node, and graph attributes which attempts to completely copy all of the data and references.

This is in contrast to the similar `D=DiGraph(G)` which returns a shallow copy of the data.

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

Examples

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

If already directed, return a (deep) copy

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

networkx.Graph.subgraph

subgraph (*nbunch*)

Return the subgraph induced on nodes in *nbunch*.

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

Parameters *nbunch* : list, iterable

A container of nodes which will be iterated through once.

Returns *G* : Graph

A subgraph of the graph with the same edge attributes.

Notes

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

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

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

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

Examples

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

3.2.2 DiGraph - Directed graphs with self loops

Overview

DiGraph (*data=None, name=""*, ***attr*)

Base class for directed graphs.

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

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

Nodes can be arbitrary (hashable) Python objects with optional key/value attributes.

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

Parameters **data** : input graph

Data to initialize graph. If data=None (default) an empty graph is created. The data can be an edge list, or any NetworkX graph object. If the corresponding optional Python packages are installed the data can also be a NumPy matrix or 2d ndarray, a SciPy sparse matrix, or a PyGraphviz graph.

name : string, optional (default='')

An optional name for the graph.

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

Attributes to add to graph as key=value pairs.

See Also:

[Graph](#), [MultiGraph](#), [MultiDiGraph](#)

Examples

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

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

G can be grown in several ways.

Nodes:

Add one node at a time:

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

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

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

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

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

Edges:

G can also be grown by adding edges.

Add one edge,

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

a list of edges,

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

or a collection of edges,

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

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

Attributes:

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

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

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

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

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

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

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

```
>>> G[1][2]['weight'] = 4.7
>>> G.edge[1][2]['weight'] = 4
```

Shortcuts:

Many common graph features allow python syntax to speed reporting.

```
>>> 1 in G      # check if node in graph
True
>>> print [n for n in G if n<3]  # iterate through nodes
[1, 2]
>>> print len(G)  # number of nodes in graph
5
>>> print G[1] # adjacency dict keyed by neighbor to edge attributes
...          # Note: you should not change this dict manually!
{2: {'color': 'blue', 'weight': 4}}
```

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

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

Reporting:

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

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

Adding and removing nodes and edges

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

networkx.DiGraph.__init__

`__init__` (*data=None*, *name=""*, ***attr*)

Initialize a graph with edges, name, graph attributes.

Parameters *data* : input graph

Data to initialize graph. If *data=None* (default) an empty graph is created. The data can be an edge list, or any NetworkX graph object. If the corresponding optional Python packages are installed the data can also be a NumPy matrix or 2d ndarray, a SciPy sparse matrix, or a PyGraphviz graph.

name : string, optional (default='')

An optional name for the graph.

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

Attributes to add to graph as key=value pairs.

See Also:

[convert](#)

Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G = nx.Graph(name='my graph')
>>> e = [(1,2), (2,3), (3,4)] # list of edges
>>> G = nx.Graph(e)
```

Arbitrary graph attribute pairs (key=value) may be assigned

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

networkx.DiGraph.add_node

`add_node` (*n*, *attr_dict=None*, ***attr*)

Add a single node *n* and update node attributes.

Parameters *n* : node

A node can be any hashable Python object except None.

attr_dict : dictionary, optional (default= no attributes)

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

attr : keyword arguments, optional

Set or change attributes using key=value.

See Also:

[add_nodes_from](#)

Notes

A hashable object is one that can be used as a key in a Python dictionary. This includes strings, numbers, tuples of strings and numbers, etc.

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

Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_node(1)
>>> G.add_node('Hello')
>>> K3 = nx.Graph([(0,1), (1,2), (2,0)])
>>> G.add_node(K3)
>>> G.number_of_nodes()
3
```

Use keywords set/change node attributes:

```
>>> G.add_node(1, size=10)
>>> G.add_node(3, weight=0.4, UTM=('13S', 382871, 3972649))
```

networkx.DiGraph.add_nodes_from

add_nodes_from(nodes, **attr)

Add multiple nodes.

Parameters nodes : iterable container

A container of nodes (list, dict, set, etc.). OR A container of (node, attribute dict) tuples. Node attributes are updated using the attribute dict.

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

Update attributes for all nodes in nodes. Node attributes specified in nodes as a tuple take precedence over attributes specified generally.

See Also:

[add_node](#)

Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_nodes_from('Hello')
>>> K3 = nx.Graph([(0,1), (1,2), (2,0)])
>>> G.add_nodes_from(K3)
>>> sorted(G.nodes())
[0, 1, 2, 'H', 'e', 'l', 'o']
```

Use keywords to update specific node attributes for every node.

```
>>> G.add_nodes_from([1,2], size=10)
>>> G.add_nodes_from([3,4], weight=0.4)
```

Use (node, attrdict) tuples to update attributes for specific nodes.

```
>>> G.add_nodes_from([(1,dict(size=11)), (2,{'color':'blue'})])
>>> G.node[1]['size']
11
>>> H = nx.Graph()
>>> H.add_nodes_from(G.nodes(data=True))
>>> H.node[1]['size']
11
```

networkx.DiGraph.remove_node

remove_node(n)

Remove node n.

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

Parameters n : node

A node in the graph

Raises NetworkXError :

If n is not in the graph.

See Also:

`remove_nodes_from`

Examples

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

networkx.DiGraph.remove_nodes_from

remove_nodes_from(nbunch)

Remove multiple nodes.

Parameters nodes : iterable container

A container of nodes (list, dict, set, etc.). If a node in the container is not in the graph it is silently ignored.

See Also:

`remove_node`

Examples

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

networkx.DiGraph.add_edge

add_edge (*u*, *v*, *attr_dict=None*, ***attr*)

Add an edge between *u* and *v*.

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

Edge attributes can be specified with keywords or by providing a dictionary with key/value pairs. See examples below.

Parameters *u,v* : nodes

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

attr_dict : dictionary, optional (default= no attributes)

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

attr : keyword arguments, optional

Edge data (or labels or objects) can be assigned using keyword arguments.

See Also:

add_edges_from add a collection of edges

Notes

Adding an edge that already exists updates the edge data.

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

Examples

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

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> e = (1,2)
>>> G.add_edge(1, 2) # explicit two-node form
>>> G.add_edge(*e) # single edge as tuple of two nodes
>>> G.add_edges_from( [(1,2)] ) # add edges from iterable container
```

Associate data to edges using keywords:

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

networkx.DiGraph.add_edges_from

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

Add all the edges in ebunch.

Parameters **ebunch** : container of edges

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

attr_dict : dictionary, optional (default= no attributes)

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

attr : keyword arguments, optional

Edge data (or labels or objects) can be assigned using keyword arguments.

See Also:

[add_edge](#) add a single edge

[add_weighted_edges_from](#) convenient way to add weighted edges

Notes

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

Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edges_from([(0,1), (1,2)]) # using a list of edge tuples
>>> e = zip(range(0,3), range(1,4))
>>> G.add_edges_from(e) # Add the path graph 0-1-2-3
```

Associate data to edges

```
>>> G.add_edges_from([(1,2), (2,3)], weight=3)
>>> G.add_edges_from([(3,4), (1,4)], label='WN2898')
```

networkx.DiGraph.add_weighted_edges_from

add_weighted_edges_from(*ebunch*, ***attr*)

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

Parameters **ebunch** : container of edges

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

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

Edge attributes to add/update for all edges.

See Also:

`add_edge` add a single edge

`add_edges_from` add multiple edges

Notes

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

Examples

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

networkx.DiGraph.remove_edge

`remove_edge(u, v)`

Remove the edge between u and v.

Parameters u,v: nodes :

Remove the edge between nodes u and v.

Raises NetworkXError :

If there is not an edge between u and v.

See Also:

`remove_edges_from` remove a collection of edges

Examples

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

networkx.DiGraph.remove_edges_from

remove_edges_from(*ebunch*)

Remove all edges specified in ebunch.

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

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

- 2-tuples (u,v) edge between u and v.
- 3-tuples (u,v,k) where k is ignored.

See Also:

[remove_edge](#) remove a single edge

Notes

Will fail silently if an edge in ebunch is not in the graph.

Examples

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

networkx.DiGraph.add_star

add_star(*nlist*, ****attr**)

Add a star.

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

Parameters **nlist** : list

A list of nodes.

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

Attributes to add to every edge in star.

See Also:

[add_path](#), [add_cycle](#)

Examples

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

networkx.DiGraph.add_path

add_path (*nlist*, ***attr*)

Add a path.

Parameters *nlist* : list

A list of nodes. A path will be constructed from the nodes (in order) and added to the graph.

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

Attributes to add to every edge in path.

See Also:

`add_star`, `add_cycle`

Examples

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

networkx.DiGraph.add_cycle

add_cycle (*nlist*, ***attr*)

Add a cycle.

Parameters *nlist* : list

A list of nodes. A cycle will be constructed from the nodes (in order) and added to the graph.

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

Attributes to add to every edge in cycle.

See Also:

`add_path`, `add_star`

Examples

```
>>> G=nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_cycle([0,1,2,3])
>>> G.add_cycle([10,11,12],weight=7)
```

networkx.DiGraph.clear

clear ()

Remove all nodes and edges from the graph.

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

Examples

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

Iterating over nodes and edges

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

networkx.DiGraph.nodes

nodes (*data=False*)

Return a list of the nodes in the graph.

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

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

Returns **nlist** : list

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

Examples

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

networkx.DiGraph.nodes_iter

nodes_iter (*data=False*)

Return an iterator over the nodes.

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

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

Returns **niter** : iterator

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

Notes

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

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

Examples

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

networkx.DiGraph.__iter__

__iter__ ()

Iterate over the nodes. Use the expression ‘for n in G’.

Returns niter : iterator

An iterator over all nodes in the graph.

Examples

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

networkx.DiGraph.edges

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

Return a list of edges.

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

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

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

data : bool, optional (default=False)

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

Returns edge_list: list of edge tuples :

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

See Also:

`edges_iter` return an iterator over the edges

Notes

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

Examples

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


networkx.DiGraph.edges_iter

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

Return an iterator over the edges.

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

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

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

data : bool, optional (default=False)

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

Returns **edge_iter** : iterator

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

See Also:

edges return a list of edges

Notes

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

Examples

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

networkx.DiGraph.out_edges

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

Return a list of edges.

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

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

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

data : bool, optional (default=False)

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

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

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

See Also:

`edges_iter` return an iterator over the edges

Notes

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

Examples

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

networkx.DiGraph.out_edges_iter

`out_edges_iter` (nbunch=None, data=False)

Return an iterator over the edges.

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

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

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

data : bool, optional (default=False)

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

Returns **edge_iter** : iterator

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

See Also:

`edges` return a list of edges

Notes

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

Examples

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

networkx.DiGraph.in_edges

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

See Also:

edges return a list of edges

networkx.DiGraph.in_edges_iter

in_edges_iter (*nbunch=None, data=False*)
Return an iterator over the incoming edges.

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

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

data : bool, optional (default=False)

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

Returns in_edge_iter : iterator

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

See Also:

edges_iter return an iterator of edges

networkx.DiGraph.get_edge_data

get_edge_data (*u, v, default=None*)
Return the attribute dictionary associated with edge (u,v).

Parameters u,v : nodes

default: any Python object (default=None) :

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

Returns edge_dict : dictionary

The edge attribute dictionary.

Notes

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

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

```

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

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

```

Examples

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

```

`networkx.DiGraph.neighbors`

neighbors(*n*)

Return a list of successor nodes of *n*.

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

`networkx.DiGraph.neighbors_iter`

neighbors_iter(*n*)

Return an iterator over successor nodes of *n*.

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

`networkx.DiGraph.__getitem__`

__getitem__(*n*)

Return a dict of neighbors of node *n*. Use the expression `'G[n]'`.

Parameters *n* : node

A node in the graph.

Returns *adj_dict* : dictionary

The adjacency dictionary for nodes connected to *n*.

Notes

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

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

Examples

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

networkx.DiGraph.successors

successors (*n*)

Return a list of successor nodes of *n*.

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

networkx.DiGraph.successors_iter

successors_iter (*n*)

Return an iterator over successor nodes of *n*.

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

networkx.DiGraph.predecessors

predecessors (*n*)

Return a list of predecessor nodes of *n*.

networkx.DiGraph.predecessors_iter

predecessors_iter (*n*)

Return an iterator over predecessor nodes of *n*.

networkx.DiGraph.adjacency_list

adjacency_list()

Return an adjacency list representation of the graph.

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

Returns `adj_list` : lists of lists

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

See Also:

`adjacency_iter`

Examples

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

networkx.DiGraph.adjacency_iter

adjacency_iter()

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

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

Returns `adj_iter` : iterator

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

See Also:

`adjacency_list`

Examples

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

networkx.DiGraph.nbunch_iter

nbunch_iter(nbunch=None)

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

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

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

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

Returns `niter` : iterator

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

Raises `NetworkXError` :

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

See Also:

`Graph.__iter__`

Notes

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

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

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

Information about graph structure

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

`networkx.DiGraph.has_node`

has_node (*n*)

Return True if the graph contains the node `n`.

Parameters `n` : node

Examples

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

It is more readable and simpler to use

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

networkx.DiGraph.__contains__

`__contains__`(*n*)

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

Examples

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

networkx.DiGraph.has_edge

`has_edge`(*u, v*)

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

Parameters *u,v* : nodes

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

Returns `edge_ind` : bool

True if edge is in the graph, False otherwise.

Examples

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

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



```
>>> G.has_edge(*e[:2]) # e is a 3-tuple (u,v,data_dictionary)
True
```

The following syntax are all equivalent:

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

networkx.DiGraph.order

order()

Return the number of nodes in the graph.

Returns nnodes : int

The number of nodes in the graph.

See Also:

`number_of_nodes`, `__len__`

networkx.DiGraph.number_of_nodes

number_of_nodes()

Return the number of nodes in the graph.

Returns nnodes : int

The number of nodes in the graph.

See Also:

`order`, `__len__`

Examples

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

networkx.DiGraph.__len__

__len__()

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

Returns nnodes : int

The number of nodes in the graph.

Examples

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

networkx.DiGraph.degree

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

Return the degree of a node or nodes.

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

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

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

weighted : bool, optional (default=False)

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

Returns **nd** : dictionary, or number

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

Examples

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

networkx.DiGraph.degree_iter

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

Return an iterator for (node, degree).

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

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

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

weighted : bool, optional (default=False)

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

Returns **nd_iter** : an iterator

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

See Also:

`degree`, `in_degree`, `out_degree`, `in_degree_iter`, `out_degree_iter`

Examples

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

networkx.DiGraph.in_degree

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

Return the in-degree of a node or nodes.

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

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

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

weighted : bool, optional (default=False)

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

Returns **nd** : dictionary, or number

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

See Also:

`degree`, `out_degree`, `in_degree_iter`

Examples

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

networkx.DiGraph.in_degree_iter

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

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

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

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

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

weighted : bool, optional (default=False)

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

Returns **nd_iter** : an iterator

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

See Also:

`degree`, `in_degree`, `out_degree`, `out_degree_iter`

Examples

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

`networkx.DiGraph.out_degree`

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

Return the out-degree of a node or nodes.

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

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

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

weighted : bool, optional (default=False)

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

Returns **nd** : dictionary, or number

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

Examples

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

networkx.DiGraph.out_degree_iter

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

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

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

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

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

weighted : bool, optional (default=False)

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

Returns **nd_iter** : an iterator

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

See Also:

`degree`, `in_degree`, `out_degree`, `in_degree_iter`

Examples

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

networkx.DiGraph.size

size (*weighted=False*)

Return the number of edges.

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

If True return the sum of the edge weights.

Returns **nedges** : int

The number of edges in the graph.

See Also:

`number_of_edges`

Examples

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

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge('a', 'b', weight=2)
>>> G.add_edge('b', 'c', weight=4)
>>> G.size()
2
>>> G.size(weighted=True)
6
```

networkx.DiGraph.number_of_edges

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

Return the number of edges between two nodes.

Parameters *u,v* : nodes, optional (default=all edges)

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

Returns *nedges* : int

The number of edges in the graph. If nodes *u* and *v* are specified return the number of edges between those nodes.

See Also:

`size`

Examples

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

networkx.DiGraph.nodes_with_selfloops

nodes_with_selfloops ()

Return a list of nodes with self loops.

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

Returns *nodelist* : list

A list of nodes with self loops.

See Also:

`selfloop_edges`, `number_of_selfloops`

Examples

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

networkx.DiGraph.selfloop_edges

selfloop_edges (*data=False*)

Return a list of selfloop edges.

A selfloop edge has the same node at both ends.

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

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

Returns **edgelist** : list of edge tuples

A list of all selfloop edges.

See Also:

`selfloop_nodes`, `number_of_selfloops`

Examples

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

networkx.DiGraph.number_of_selfloops

number_of_selfloops ()

Return the number of selfloop edges.

A selfloop edge has the same node at both ends.

Returns **nloops** : int

The number of selfloops.

See Also:

`selfloop_nodes`, `selfloop_edges`

Examples

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

Making copies and subgraphs

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

networkx.DiGraph.copy

`copy()`

Return a copy of the graph.

Returns `G` : Graph

A copy of the graph.

See Also:

`to_directed` return a directed copy of the graph.

Notes

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

Examples

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

networkx.DiGraph.to_undirected

`to_undirected()`

Return an undirected representation of the digraph.

Returns `G` : Graph

An undirected graph with the same name and nodes and with edge (u,v,data) if either (u,v,data) or (v,u,data) is in the digraph. If both edges exist in digraph and their edge data is different, only one edge is created with an arbitrary choice of which edge data to use. You must check and correct for this manually if desired.

Notes

If edges in both directions (u,v) and (v,u) exist in the graph, attributes for the new undirected edge will be a combination of the attributes of the directed edges. The edge data is updated in the (arbitrary) order that the edges are encountered. For more customized control of the edge attributes use `add_edge()`.

This returns a “deepcopy” of the edge, node, and graph attributes which attempts to completely copy all of the data and references.

This is in contrast to the similar `G=DiGraph(D)` which returns a shallow copy of the data.

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

`networkx.DiGraph.to_directed`

`to_directed()`

Return a directed copy of the graph.

Returns `G` : `DiGraph`

A deepcopy of the graph.

Notes

This returns a “deepcopy” of the edge, node, and graph attributes which attempts to completely copy all of the data and references.

This is in contrast to the similar `D=DiGraph(G)` which returns a shallow copy of the data.

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

Examples

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

If already directed, return a (deep) copy

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

networkx.DiGraph.subgraph

subgraph (*nbunch*)

Return the subgraph induced on nodes in nbunch.

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

Parameters **nbunch** : list, iterable

A container of nodes which will be iterated through once.

Returns **G** : Graph

A subgraph of the graph with the same edge attributes.

Notes

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

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

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

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

Examples

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

networkx.DiGraph.reverse

reverse (*copy=True*)

Return the reverse of the graph.

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

Parameters **copy** : bool optional (default=True)

If True, return a new DiGraph holding the reversed edges. If False, reverse the reverse graph is created using the original graph (this changes the original graph).

3.2.3 MultiGraph - Undirected graphs with self loops and parallel edges

Overview

MultiGraph (*data=None, name="", **attr*)

An undirected graph class that can store multiedges.

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

A MultiGraph holds undirected edges. Self loops are allowed.

Nodes can be arbitrary (hashable) Python objects with optional key/value attributes.

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

Parameters `data` : input graph

Data to initialize graph. If `data=None` (default) an empty graph is created. The data can be an edge list, or any NetworkX graph object. If the corresponding optional Python packages are installed the data can also be a NumPy matrix or 2d ndarray, a SciPy sparse matrix, or a PyGraphviz graph.

name : string, optional (default='')

An optional name for the graph.

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

Attributes to add to graph as key=value pairs.

See Also:

[Graph](#), [DiGraph](#), [MultiDiGraph](#)

Examples

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

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

G can be grown in several ways.

Nodes:

Add one node at a time:

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

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

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

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

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

Edges:

G can also be grown by adding edges.

Add one edge,

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

a list of edges,

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

or a collection of edges,

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

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

```
>>> G.add_edges_from([(4,5,dict(route=282)), (4,5,dict(route=37))])
>>> G[4]
{3: {0: {}}, 5: {0: {}, 1: {'route': 282}, 2: {'route': 37}}}
```

Attributes:

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

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

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

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

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

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

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

Shortcuts:

Many common graph features allow python syntax to speed reporting.

```
>>> 1 in G      # check if node in graph
True
>>> print [n for n in G if n<3]  # iterate through nodes
[1, 2]
>>> print len(G)  # number of nodes in graph
5
>>> print G[1]  # adjacency dict keyed by neighbor to edge attributes
```

```
...          # Note: you should not change this dict manually!
{2: {0: {'weight': 4}, 1: {'color': 'blue'}}}
```

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

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

Reporting:

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

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

Adding and removing nodes and edges

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

networkx.MultiGraph.__init__

`__init__(data=None, name="", **attr)`
Initialize a graph with edges, name, graph attributes.

Parameters `data` : input graph

Data to initialize graph. If `data=None` (default) an empty graph is created. The data can be an edge list, or any NetworkX graph object. If the corresponding optional Python

packages are installed the data can also be a NumPy matrix or 2d ndarray, a SciPy sparse matrix, or a PyGraphviz graph.

name : string, optional (default='')

An optional name for the graph.

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

Attributes to add to graph as key=value pairs.

See Also:

`convert`

Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G = nx.Graph(name='my graph')
>>> e = [(1,2), (2,3), (3,4)] # list of edges
>>> G = nx.Graph(e)
```

Arbitrary graph attribute pairs (key=value) may be assigned

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

networkx.MultiGraph.add_node

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

Add a single node *n* and update node attributes.

Parameters **n** : node

A node can be any hashable Python object except None.

attr_dict : dictionary, optional (default= no attributes)

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

attr : keyword arguments, optional

Set or change attributes using key=value.

See Also:

`add_nodes_from`

Notes

A hashable object is one that can be used as a key in a Python dictionary. This includes strings, numbers, tuples of strings and numbers, etc.

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

Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_node(1)
>>> G.add_node('Hello')
>>> K3 = nx.Graph([(0,1), (1,2), (2,0)])
>>> G.add_node(K3)
>>> G.number_of_nodes()
3
```

Use keywords set/change node attributes:

```
>>> G.add_node(1, size=10)
>>> G.add_node(3, weight=0.4, UTM=('13S', 382871, 3972649))
```

networkx.MultiGraph.add_nodes_from

add_nodes_from(nodes, **attr)

Add multiple nodes.

Parameters nodes : iterable container

A container of nodes (list, dict, set, etc.). OR A container of (node, attribute dict) tuples. Node attributes are updated using the attribute dict.

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

Update attributes for all nodes in nodes. Node attributes specified in nodes as a tuple take precedence over attributes specified generally.

See Also:

`add_node`

Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_nodes_from('Hello')
>>> K3 = nx.Graph([(0,1), (1,2), (2,0)])
>>> G.add_nodes_from(K3)
>>> sorted(G.nodes())
[0, 1, 2, 'H', 'e', 'l', 'o']
```

Use keywords to update specific node attributes for every node.

```
>>> G.add_nodes_from([1,2], size=10)
>>> G.add_nodes_from([3,4], weight=0.4)
```

Use (node, attrdict) tuples to update attributes for specific nodes.

```
>>> G.add_nodes_from([(1,dict(size=11)), (2,{'color':'blue'})])
>>> G.node[1]['size']
11
>>> H = nx.Graph()
```

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

networkx.MultiGraph.remove_node

remove_node(*n*)

Remove node *n*.

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

Parameters *n* : node

A node in the graph

Raises `NetworkXError` :

If *n* is not in the graph.

See Also:

`remove_nodes_from`

Examples

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

networkx.MultiGraph.remove_nodes_from

remove_nodes_from(*nodes*)

Remove multiple nodes.

Parameters *nodes* : iterable container

A container of nodes (list, dict, set, etc.). If a node in the container is not in the graph it is silently ignored.

See Also:

`remove_node`

Examples

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



```
[0, 1, 2]
>>> G.remove_nodes_from(e)
>>> G.nodes()
[]
```

networkx.MultiGraph.add_edge

add_edge (*u*, *v*, *key=None*, *attr_dict=None*, ***attr*)

Add an edge between *u* and *v*.

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

Edge attributes can be specified with keywords or by providing a dictionary with key/value pairs. See examples below.

Parameters *u,v* : nodes

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

key : hashable identifier, optional (default=lowest unused integer)

Used to distinguish multiedges between a pair of nodes.

attr_dict : dictionary, optional (default= no attributes)

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

attr : keyword arguments, optional

Edge data (or labels or objects) can be assigned using keyword arguments.

See Also:

add_edges_from add a collection of edges

Notes

To replace/update edge data, use the optional *key* argument to identify a unique edge. Otherwise a new edge will be created.

NetworkX algorithms designed for weighted graphs cannot use multigraphs directly because it is not clear how to handle multiedge weights. Convert to Graph using edge attribute 'weight' to enable weighted graph algorithms.

Examples

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

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> e = (1,2)
>>> G.add_edge(1, 2) # explicit two-node form
>>> G.add_edge(*e) # single edge as tuple of two nodes
>>> G.add_edges_from( [(1,2)] ) # add edges from iterable container
```

Associate data to edges using keywords:

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

networkx.MultiGraph.add_edges_from

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

Add all the edges in *ebunch*.

Parameters *ebunch* : container of edges

Each edge given in the container will be added to the graph. The edges can be:

- 2-tuples (u,v) or
- 3-tuples (u,v,d) for an edge attribute dict d, or
- 4-tuples (u,v,k,d) for an edge identified by key k

attr_dict : dictionary, optional (default= no attributes)

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

attr : keyword arguments, optional

Edge data (or labels or objects) can be assigned using keyword arguments.

See Also:

add_edge add a single edge

add_weighted_edges_from convenient way to add weighted edges

Notes

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

Examples

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edges_from([(0,1), (1,2)]) # using a list of edge tuples
>>> e = zip(range(0,3), range(1,4))
>>> G.add_edges_from(e) # Add the path graph 0-1-2-3
```

Associate data to edges

```
>>> G.add_edges_from([(1,2), (2,3)], weight=3)
>>> G.add_edges_from([(3,4), (1,4)], label='WN2898')
```

networkx.MultiGraph.add_weighted_edges_from

add_weighted_edges_from (*ebunch*, ***attr*)

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

Parameters *ebunch* : container of edges

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

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

Edge attributes to add/update for all edges.

See Also:

[add_edge](#) add a single edge

[add_edges_from](#) add multiple edges

Notes

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

Examples

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

networkx.MultiGraph.remove_edge

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

Remove an edge between *u* and *v*.

Parameters *u,v: nodes* :

Remove an edge between nodes *u* and *v*.

key : hashable identifier, optional (default=None)

Used to distinguish multiple edges between a pair of nodes. If None remove a single (arbitrary) edge between *u* and *v*.

Raises **NetworkXError** :

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

See Also:

[remove_edges_from](#) remove a collection of edges

Examples

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

For multiple edges

```
>>> G = nx.MultiGraph() # or MultiDiGraph, etc
>>> G.add_edges_from([(1,2), (1,2), (1,2)])
>>> G.remove_edge(1,2) # remove a single (arbitrary) edge
```

For edges with keys

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

networkx.MultiGraph.remove_edges_from

remove_edges_from(*ebunch*)

Remove all edges specified in ebunch.

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

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

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

See Also:

[remove_edge](#) remove a single edge

Notes

Will fail silently if an edge in ebunch is not in the graph.

Examples

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

Removing multiple copies of edges

```

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

```

networkx.MultiGraph.add_star

add_star (*nlist*, ***attr*)

Add a star.

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

Parameters *nlist* : list

A list of nodes.

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

Attributes to add to every edge in star.

See Also:

`add_path`, `add_cycle`

Examples

```

>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_star([0,1,2,3])
>>> G.add_star([10,11,12],weight=2)

```

networkx.MultiGraph.add_path

add_path (*nlist*, ***attr*)

Add a path.

Parameters *nlist* : list

A list of nodes. A path will be constructed from the nodes (in order) and added to the graph.

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

Attributes to add to every edge in path.

See Also:

`add_star`, `add_cycle`

Examples

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

networkx.MultiGraph.add_cycle

add_cycle (*nlist*, ***attr*)

Add a cycle.

Parameters *nlist* : list

A list of nodes. A cycle will be constructed from the nodes (in order) and added to the graph.

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

Attributes to add to every edge in cycle.

See Also:

`add_path`, `add_star`

Examples

```
>>> G=nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_cycle([0,1,2,3])
>>> G.add_cycle([10,11,12],weight=7)
```

networkx.MultiGraph.clear

clear ()

Remove all nodes and edges from the graph.

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

Examples

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

Iterating over nodes and edges

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

networkx.MultiGraph.nodes

nodes (*data=False*)

Return a list of the nodes in the graph.

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

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

Returns **nlist** : list

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

Examples

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

networkx.MultiGraph.nodes_iter

nodes_iter (*data=False*)

Return an iterator over the nodes.

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

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

Returns **niter** : iterator

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

Notes

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

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

Examples

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

networkx.MultiGraph.__iter__

`__iter__()`

Iterate over the nodes. Use the expression ‘for n in G’.

Returns niter : iterator

An iterator over all nodes in the graph.

Examples

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

networkx.MultiGraph.edges

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

Return a list of edges.

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

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

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

data : bool, optional (default=False)

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

keys : bool, optional (default=False)

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

Returns edge_list: list of edge tuples :

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

See Also:

`edges_iter` return an iterator over the edges

Notes

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

Examples

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

networkx.MultiGraph.edges_iter

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

Return an iterator over the edges.

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

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

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

data : bool, optional (default=False)

If True, return edge attribute dict with each edge.

keys : bool, optional (default=False)

If True, return edge keys with each edge.

Returns `edge_iter` : iterator

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

See Also:

`edges` return a list of edges

Notes

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

Examples

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

networkx.MultiGraph.get_edge_data

get_edge_data (u, v, key=None, default=None)

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

Parameters `u,v` : nodes

default : any Python object (default=None) :

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

key : hashable identifier, optional (default=None)

Return data only for the edge with specified key.

Returns `edge_dict` : dictionary

The edge attribute dictionary.

Notes

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

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

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

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

Examples

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

networkx.MultiGraph.neighbors

`neighbors(n)`

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

Parameters `n`: node

A node in the graph

Returns `nlist`: list

A list of nodes that are adjacent to `n`.

Raises `NetworkXError`:

If the node `n` is not in the graph.

Notes

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

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

Examples

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

networkx.MultiGraph.neighbors_iter

neighbors_iter(*n*)

Return an iterator over all neighbors of node *n*.

Notes

It is faster to use the idiom “in G[0]”, e.g. >>> for n in G[0]: ... print n 1

Examples

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

networkx.MultiGraph.__getitem__

__getitem__(*n*)

Return a dict of neighbors of node *n*. Use the expression ‘G[*n*]’.

Parameters *n* : node

A node in the graph.

Returns *adj_dict* : dictionary

The adjacency dictionary for nodes connected to *n*.

Notes

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

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

Examples

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

networkx.MultiGraph.adjacency_list

adjacency_list()

Return an adjacency list representation of the graph.

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

Returns `adj_list` : lists of lists

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

See Also:

`adjacency_iter`

Examples

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

networkx.MultiGraph.adjacency_iter

adjacency_iter()

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

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

Returns `adj_iter` : iterator

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

See Also:

`adjacency_list`

Examples

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

networkx.MultiGraph.nbunch_iter

nbunch_iter(nbunch=None)

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

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

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

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

Returns `niter` : iterator

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

Raises `NetworkXError` :

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

See Also:

`Graph.__iter__`

Notes

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

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

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

Information about graph structure

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

networkx.MultiGraph.has_node

has_node (*n*)

Return True if the graph contains the node `n`.

Parameters `n` : node

Examples

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

It is more readable and simpler to use

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

networkx.MultiGraph.__contains__

__contains__(*n*)

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

Examples

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

networkx.MultiGraph.has_edge

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

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

Parameters *u,v* : nodes

Nodes can be, for example, strings or numbers.

key : hashable identifier, optional (default=None)

If specified return True only if the edge with *key* is found.

Returns *edge_ind* : bool

True if edge is in the graph, False otherwise.

Examples

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

```
>>> G = nx.MultiGraph() # or MultiDiGraph
>>> G.add_path([0,1,2,3])
>>> G.has_edge(0,1) # using two nodes
True
>>> e = (0,1)
>>> G.has_edge(*e) # e is a 2-tuple (u,v)
True
>>> G.add_edge(0,1,key='a')
>>> G.has_edge(0,1,key='a') # specify key
True
>>> e=(0,1,'a')
>>> G.has_edge(*e) # e is a 3-tuple (u,v,'a')
True
```

The following syntax are equivalent:

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

networkx.MultiGraph.order

order()

Return the number of nodes in the graph.

Returns `nnodes` : int

The number of nodes in the graph.

See Also:

`number_of_nodes`, `__len__`

networkx.MultiGraph.number_of_nodes

number_of_nodes()

Return the number of nodes in the graph.

Returns `nnodes` : int

The number of nodes in the graph.

See Also:

`order`, `__len__`

Examples

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

networkx.MultiGraph.__len__

__len__()

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

Returns `nnodes` : int

The number of nodes in the graph.

Examples


```

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

```

networkx.MultiGraph.degree

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

Return the degree of a node or nodes.

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

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

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

weighted : bool, optional (default=False)

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

Returns **nd** : dictionary, or number

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

Examples

```

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

```

networkx.MultiGraph.degree_iter

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

Return an iterator for (node, degree).

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

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

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

weighted : bool, optional (default=False)

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

Returns **nd_iter** : an iterator

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

See Also:

[degree](#)

Examples

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

networkx.MultiGraph.size

size (*weighted=False*)

Return the number of edges.

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

If True return the sum of the edge weights.

Returns **nedges** : int

The number of edges in the graph.

See Also:

`number_of_edges`

Examples

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

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge('a', 'b', weight=2)
>>> G.add_edge('b', 'c', weight=4)
>>> G.size()
2
>>> G.size(weighted=True)
6
```

networkx.MultiGraph.number_of_edges

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

Return the number of edges between two nodes.

Parameters **u,v** : nodes, optional (default=all edges)

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

Returns **nedges** : int

The number of edges in the graph. If nodes *u* and *v* are specified return the number of edges between those nodes.

See Also:

`size`

Examples

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

networkx.MultiGraph.nodes_with_selfloops

`nodes_with_selfloops()`

Return a list of nodes with self loops.

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

Returns `nodelist` : list

A list of nodes with self loops.

See Also:

`selfloop_edges`, `number_of_selfloops`

Examples

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

networkx.MultiGraph.selfloop_edges

`selfloop_edges(data=False, keys=False)`

Return a list of selfloop edges.

A selfloop edge has the same node at both ends.

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

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

keys : bool, optional (default=False)

If True, return edge keys with each edge.

Returns edgelist : list of edge tuples

A list of all selfloop edges.

See Also:

`selfloop_nodes`, `number_of_selfloops`

Examples

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

`networkx.MultiGraph.number_of_selfloops`

number_of_selfloops ()

Return the number of selfloop edges.

A selfloop edge has the same node at both ends.

Returns nloops : int

The number of selfloops.

See Also:

`selfloop_nodes`, `selfloop_edges`

Examples

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

Making copies and subgraphs

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

networkx.MultiGraph.copy

`copy()`

Return a copy of the graph.

Returns `G` : Graph

A copy of the graph.

See Also:

`to_directed` return a directed copy of the graph.

Notes

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

Examples

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

networkx.MultiGraph.to_undirected

`to_undirected()`

Return an undirected copy of the graph.

Returns `G` : Graph/MultiGraph

A deepcopy of the graph.

See Also:

`copy`, `add_edge`, `add_edges_from`

Notes

This returns a “deepcopy” of the edge, node, and graph attributes which attempts to completely copy all of the data and references.

This is in contrast to the similar `G=DiGraph(D)` which returns a shallow copy of the data.

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

Examples

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

networkx.MultiGraph.to_directed

to_directed()

Return a directed representation of the graph.

Returns **G** : MultiDiGraph

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

Notes

This returns a “deepcopy” of the edge, node, and graph attributes which attempts to completely copy all of the data and references.

This is in contrast to the similar `D=DiGraph(G)` which returns a shallow copy of the data.

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

Examples

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

If already directed, return a (deep) copy

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

networkx.MultiGraph.subgraph

subgraph (*nbunch*)

Return the subgraph induced on nodes in nbunch.

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

Parameters **nbunch** : list, iterable

A container of nodes which will be iterated through once.

Returns **G** : Graph

A subgraph of the graph with the same edge attributes.

Notes

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

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

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

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

Examples

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

3.2.4 MultiDiGraph - Directed graphs with self loops and parallel edges

Overview

MultiDiGraph (*data=None, name="", **attr*)

A directed graph class that can store multiedges.

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

A MultiDiGraph holds directed edges. Self loops are allowed.

Nodes can be arbitrary (hashable) Python objects with optional key/value attributes.

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

Parameters **data** : input graph

Data to initialize graph. If `data=None` (default) an empty graph is created. The data can be an edge list, or any NetworkX graph object. If the corresponding optional Python packages are installed the data can also be a NumPy matrix or 2d ndarray, a SciPy sparse matrix, or a PyGraphviz graph.

name : string, optional (default='')

An optional name for the graph.

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

Attributes to add to graph as key=value pairs.

See Also:

`Graph`, `DiGraph`, `MultiGraph`

Examples

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

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

G can be grown in several ways.

Nodes:

Add one node at a time:

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

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

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

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

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

Edges:

G can also be grown by adding edges.

Add one edge,

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

a list of edges,

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

or a collection of edges,

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

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

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


Attributes:

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

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

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

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

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

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

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

Shortcuts:

Many common graph features allow python syntax to speed reporting.

```
>>> 1 in G      # check if node in graph
True
>>> print [n for n in G if n<3]  # iterate through nodes
[1, 2]
>>> print len(G)  # number of nodes in graph
5
>>> print G[1] # adjacency dict keyed by neighbor to edge attributes
...          # Note: you should not change this dict manually!
{2: {0: {'weight': 4}, 1: {'color': 'blue'}}}
```

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

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

Reporting:

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

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

Adding and Removing Nodes and Edges

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

networkx.MultiDiGraph.__init__

`__init__ (data=None, name="", **attr)`
 Initialize a graph with edges, name, graph attributes.

Parameters `data` : input graph

Data to initialize graph. If data=None (default) an empty graph is created. The data can be an edge list, or any NetworkX graph object. If the corresponding optional Python packages are installed the data can also be a NumPy matrix or 2d ndarray, a SciPy sparse matrix, or a PyGraphviz graph.

name : string, optional (default='')

An optional name for the graph.

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

Attributes to add to graph as key=value pairs.

See Also:

convert

Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G = nx.Graph(name='my graph')
>>> e = [(1,2), (2,3), (3,4)] # list of edges
>>> G = nx.Graph(e)
```

Arbitrary graph attribute pairs (key=value) may be assigned

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

networkx.MultiDiGraph.add_node

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

Add a single node *n* and update node attributes.

Parameters *n* : node

A node can be any hashable Python object except None.

attr_dict : dictionary, optional (default= no attributes)

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

attr : keyword arguments, optional

Set or change attributes using key=value.

See Also:

[add_nodes_from](#)

Notes

A hashable object is one that can be used as a key in a Python dictionary. This includes strings, numbers, tuples of strings and numbers, etc.

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

Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_node(1)
>>> G.add_node('Hello')
>>> K3 = nx.Graph([(0,1), (1,2), (2,0)])
>>> G.add_node(K3)
>>> G.number_of_nodes()
3
```

Use keywords set/change node attributes:

```
>>> G.add_node(1, size=10)
>>> G.add_node(3, weight=0.4, UTM=('13S', 382871, 3972649))
```

networkx.MultiDiGraph.add_nodes_from

add_nodes_from(nodes, **attr)

Add multiple nodes.

Parameters nodes : iterable container

A container of nodes (list, dict, set, etc.). OR A container of (node, attribute dict) tuples.
Node attributes are updated using the attribute dict.

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

Update attributes for all nodes in nodes. Node attributes specified in nodes as a tuple take precedence over attributes specified generally.

See Also:

[add_node](#)

Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_nodes_from('Hello')
>>> K3 = nx.Graph([(0,1), (1,2), (2,0)])
>>> G.add_nodes_from(K3)
>>> sorted(G.nodes())
[0, 1, 2, 'H', 'e', 'l', 'o']
```

Use keywords to update specific node attributes for every node.

```
>>> G.add_nodes_from([1,2], size=10)
>>> G.add_nodes_from([3,4], weight=0.4)
```

Use (node, attrdict) tuples to update attributes for specific nodes.

```
>>> G.add_nodes_from([(1,dict(size=11)), (2,{'color':'blue'})])
>>> G.node[1]['size']
11
>>> H = nx.Graph()
>>> H.add_nodes_from(G.nodes(data=True))
>>> H.node[1]['size']
11
```

networkx.MultiDiGraph.remove_node

remove_node(n)

Remove node n.

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

Parameters n : node

A node in the graph

Raises `NetworkXError` :

If `n` is not in the graph.

See Also:

`remove_nodes_from`

Examples

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

`networkx.MultiDiGraph.remove_nodes_from`

`remove_nodes_from` (*nbunch*)

Remove multiple nodes.

Parameters `nodes` : iterable container

A container of nodes (list, dict, set, etc.). If a node in the container is not in the graph it is silently ignored.

See Also:

`remove_node`

Examples

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

`networkx.MultiDiGraph.add_edge`

`add_edge` (*u, v, key=None, attr_dict=None, **attr*)

Add an edge between `u` and `v`.

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

Edge attributes can be specified with keywords or by providing a dictionary with key/value pairs. See examples below.

Parameters *u,v* : nodes

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

key : hashable identifier, optional (default=lowest unused integer)

Used to distinguish multiedges between a pair of nodes.

attr_dict : dictionary, optional (default= no attributes)

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

attr : keyword arguments, optional

Edge data (or labels or objects) can be assigned using keyword arguments.

See Also:

[add_edges_from](#) add a collection of edges

Notes

To replace/update edge data, use the optional key argument to identify a unique edge. Otherwise a new edge will be created.

NetworkX algorithms designed for weighted graphs cannot use multigraphs directly because it is not clear how to handle multiedge weights. Convert to Graph using edge attribute 'weight' to enable weighted graph algorithms.

Examples

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

```
>>> G = nx.MultiDiGraph()
>>> e = (1,2)
>>> G.add_edge(1, 2)           # explicit two-node form
>>> G.add_edge(*e)           # single edge as tuple of two nodes
>>> G.add_edges_from( [(1,2)] ) # add edges from iterable container
```

Associate data to edges using keywords:

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

networkx.MultiDiGraph.add_edges_from

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

Add all the edges in ebunch.

Parameters *ebunch* : container of edges

Each edge given in the container will be added to the graph. The edges can be:

- 2-tuples (u,v) or

- 3-tuples (u,v,d) for an edge attribute dict d, or
- 4-tuples (u,v,k,d) for an edge identified by key k

attr_dict : dictionary, optional (default= no attributes)

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

attr : keyword arguments, optional

Edge data (or labels or objects) can be assigned using keyword arguments.

See Also:

[add_edge](#) add a single edge

[add_weighted_edges_from](#) convenient way to add weighted edges

Notes

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

Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edges_from([(0,1), (1,2)]) # using a list of edge tuples
>>> e = zip(range(0,3), range(1,4))
>>> G.add_edges_from(e) # Add the path graph 0-1-2-3
```

Associate data to edges

```
>>> G.add_edges_from([(1,2), (2,3)], weight=3)
>>> G.add_edges_from([(3,4), (1,4)], label='WN2898')
```

networkx.MultiDiGraph.add_weighted_edges_from

add_weighted_edges_from(*ebunch*, ****attr**)

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

Parameters **ebunch** : container of edges

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

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

Edge attributes to add/update for all edges.

See Also:

[add_edge](#) add a single edge

[add_edges_from](#) add multiple edges

Notes

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

Examples

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

networkx.MultiDiGraph.remove_edge

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

Remove an edge between *u* and *v*.

Parameters *u,v: nodes* :

Remove an edge between nodes *u* and *v*.

key : hashable identifier, optional (default=None)

Used to distinguish multiple edges between a pair of nodes. If None remove a single (arbitrary) edge between *u* and *v*.

Raises **NetworkXError** :

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

See Also:

[remove_edges_from](#) remove a collection of edges

Examples

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

For multiple edges

```
>>> G = nx.MultiDiGraph()
>>> G.add_edges_from([(1,2), (1,2), (1,2)])
>>> G.remove_edge(1,2) # remove a single (arbitrary) edge
```

For edges with keys

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


networkx.MultiDiGraph.remove_edges_from

remove_edges_from(*ebunch*)

Remove all edges specified in ebunch.

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

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

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

See Also:

[remove_edge](#) remove a single edge

Notes

Will fail silently if an edge in ebunch is not in the graph.

Examples

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

Removing multiple copies of edges

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

networkx.MultiDiGraph.add_star

add_star(*nlist*, ***attr*)

Add a star.

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

Parameters **nlist** : list

A list of nodes.

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

Attributes to add to every edge in star.

See Also:

`add_path`, `add_cycle`

Examples

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

`networkx.MultiDiGraph.add_path`

add_path (*nlist*, ***attr*)

Add a path.

Parameters *nlist* : list

A list of nodes. A path will be constructed from the nodes (in order) and added to the graph.

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

Attributes to add to every edge in path.

See Also:

`add_star`, `add_cycle`

Examples

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

`networkx.MultiDiGraph.add_cycle`

add_cycle (*nlist*, ***attr*)

Add a cycle.

Parameters *nlist* : list

A list of nodes. A cycle will be constructed from the nodes (in order) and added to the graph.

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

Attributes to add to every edge in cycle.

See Also:

`add_path`, `add_star`

Examples

```
>>> G=nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_cycle([0,1,2,3])
>>> G.add_cycle([10,11,12],weight=7)
```

networkx.MultiDiGraph.clear

clear()

Remove all nodes and edges from the graph.

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

Examples

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

Iterating over nodes and edges

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

networkx.MultiDiGraph.nodes

nodes (*data=False*)

Return a list of the nodes in the graph.

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

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

Returns **nlist** : list

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

Examples

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

networkx.MultiDiGraph.nodes_iter

nodes_iter (*data=False*)

Return an iterator over the nodes.

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

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

Returns **niter** : iterator

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

Notes

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

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

Examples

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

networkx.MultiDiGraph.__iter__

__iter__ ()

Iterate over the nodes. Use the expression ‘for n in G’.

Returns **niter** : iterator

An iterator over all nodes in the graph.

Examples

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

```
... print n,  
0 1 2 3
```

networkx.MultiDiGraph.edges

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

Return a list of edges.

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

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

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

data : bool, optional (default=False)

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

keys : bool, optional (default=False)

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

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

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

See Also:

[edges_iter](#) return an iterator over the edges

Notes

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

Examples

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

networkx.MultiDiGraph.edges_iter

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

Return an iterator over the edges.

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

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

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

data : bool, optional (default=False)

If True, return edge attribute dict with each edge.

keys : bool, optional (default=False)

If True, return edge keys with each edge.

Returns **edge_iter** : iterator

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

See Also:

edges return a list of edges

Notes

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

Examples

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

networkx.MultiDiGraph.out_edges

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

Return a list of edges.

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

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

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

data : bool, optional (default=False)

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

Returns `edge_list`: list of edge tuples :

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

See Also:

`edges_iter` return an iterator over the edges

Notes

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

Examples

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

networkx.MultiDiGraph.out_edges_iter

`out_edges_iter` (nbunch=None, data=False, keys=False)

Return an iterator over the edges.

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

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

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

data : bool, optional (default=False)

If True, return edge attribute dict with each edge.

keys : bool, optional (default=False)

If True, return edge keys with each edge.

Returns `edge_iter` : iterator

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

See Also:

`edges` return a list of edges

Notes

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

Examples

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

networkx.MultiDiGraph.in_edges

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

Return a list of the incoming edges.

See Also:

edges return a list of edges

networkx.MultiDiGraph.in_edges_iter

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

Return an iterator over the incoming edges.

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

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

data : bool, optional (default=False)

If True, return edge attribute dict with each edge.

keys : bool, optional (default=False)

If True, return edge keys with each edge.

Returns **in_edge_iter** : iterator

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

See Also:

edges_iter return an iterator of edges

networkx.MultiDiGraph.get_edge_data

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

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

Parameters *u,v* : nodes

default: any Python object (default=None) :

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

key : hashable identifier, optional (default=None)

Return data only for the edge with specified key.

Returns *edge_dict* : dictionary

The edge attribute dictionary.

Notes

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

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

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

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

Examples

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

networkx.MultiDiGraph.neighbors

neighbors (*n*)

Return a list of successor nodes of *n*.

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

`networkx.MultiDiGraph.neighbors_iter`

neighbors_iter (*n*)

Return an iterator over successor nodes of *n*.

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

`networkx.MultiDiGraph.__getitem__`

__getitem__ (*n*)

Return a dict of neighbors of node *n*. Use the expression '`G[n]`'.

Parameters **n** : node

A node in the graph.

Returns **adj_dict** : dictionary

The adjacency dictionary for nodes connected to *n*.

Notes

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

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

Examples

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

`networkx.MultiDiGraph.successors`

successors (*n*)

Return a list of successor nodes of *n*.

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

`networkx.MultiDiGraph.successors_iter`

successors_iter (*n*)

Return an iterator over successor nodes of *n*.

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

networkx.MultiDiGraph.predecessors

predecessors (*n*)

Return a list of predecessor nodes of *n*.

networkx.MultiDiGraph.predecessors_iter

predecessors_iter (*n*)

Return an iterator over predecessor nodes of *n*.

networkx.MultiDiGraph.adjacency_list

adjacency_list ()

Return an adjacency list representation of the graph.

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

Returns `adj_list` : lists of lists

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

See Also:

`adjacency_iter`

Examples

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

networkx.MultiDiGraph.adjacency_iter

adjacency_iter ()

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

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

Returns `adj_iter` : iterator

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

See Also:

`adjacency_list`

Examples

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

networkx.MultiDiGraph.nbunch_iter

nbunch_iter (*nbunch=None*)

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

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

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

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

Returns **niter** : iterator

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

Raises **NetworkXError** :

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

See Also:

`Graph.__iter__`

Notes

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

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

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

Information about graph structure

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

networkx.MultiDiGraph.has_node

has_node (*n*)

Return True if the graph contains the node n.

Parameters *n* : node

Examples

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

It is more readable and simpler to use

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

networkx.MultiDiGraph.__contains__

__contains__ (*n*)

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

Examples

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

networkx.MultiDiGraph.has_edge

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

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

Parameters *u,v* : nodes

Nodes can be, for example, strings or numbers.

key : hashable identifier, optional (default=None)

If specified return True only if the edge with *key* is found.

Returns *edge_ind* : bool

True if edge is in the graph, False otherwise.

Examples

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

```
>>> G = nx.MultiGraph() # or MultiDiGraph
>>> G.add_path([0,1,2,3])
>>> G.has_edge(0,1) # using two nodes
True
>>> e = (0,1)
>>> G.has_edge(*e) # e is a 2-tuple (u,v)
True
>>> G.add_edge(0,1,key='a')
>>> G.has_edge(0,1,key='a') # specify key
True
>>> e=(0,1,'a')
>>> G.has_edge(*e) # e is a 3-tuple (u,v,'a')
True
```

The following syntax are equivalent:

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

networkx.MultiDiGraph.order

order ()

Return the number of nodes in the graph.

Returns `nnodes` : int

The number of nodes in the graph.

See Also:

`number_of_nodes`, `__len__`

`networkx.MultiDiGraph.number_of_nodes`

number_of_nodes ()

Return the number of nodes in the graph.

Returns `nnodes` : int

The number of nodes in the graph.

See Also:

`order`, `__len__`

Examples

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

`networkx.MultiDiGraph.__len__`

__len__ ()

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

Returns `nnodes` : int

The number of nodes in the graph.

Examples

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

`networkx.MultiDiGraph.degree`

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

Return the degree of a node or nodes.

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

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

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

weighted : bool, optional (default=False)

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

Returns **nd** : dictionary, or number

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

Examples

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

networkx.MultiDiGraph.degree_iter

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

Return an iterator for (node, degree).

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

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

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

weighted : bool, optional (default=False)

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

Returns **nd_iter** : an iterator

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

See Also:

[degree](#)

Examples

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

networkx.MultiDiGraph.in_degree

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

Return the in-degree of a node or nodes.

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

Parameters **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

weighted : bool, optional (default=False)

If True return the sum of edge weights adjacent to the node.

Returns **nd** : dictionary, or number

A dictionary with nodes as keys and in-degree as values or a number if a single node is specified.

See Also:

[degree](#), [out_degree](#), [in_degree_iter](#)

Examples

```
>>> G = nx.DiGraph() # or MultiDiGraph
>>> G.add_path([0,1,2,3])
>>> G.in_degree(0)
0
>>> G.in_degree([0,1])
{0: 0, 1: 1}
>>> G.in_degree([0,1]).values()
[0, 1]
```

networkx.MultiDiGraph.in_degree_iter

in_degree_iter (*nbunch=None, weighted=False*)

Return an iterator for (node, in-degree).

The node in-degree is the number of edges pointing in to the node.

Parameters **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

weighted : bool, optional (default=False)

If True return the sum of edge weights adjacent to the node.

Returns **nd_iter** : an iterator

The iterator returns two-tuples of (node, in-degree).

See Also:

[degree](#), [in_degree](#), [out_degree](#), [out_degree_iter](#)

Examples

```
>>> G = nx.MultiDiGraph()
>>> G.add_path([0,1,2,3])
>>> list(G.in_degree_iter(0)) # node 0 with degree 0
[(0, 0)]
>>> list(G.in_degree_iter([0,1]))
[(0, 0), (1, 1)]
```

networkx.MultiDiGraph.out_degree

out_degree (*nbunch=None, weighted=False*)

Return the out-degree of a node or nodes.

The node out-degree is the number of edges pointing out of the node.

Parameters **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

weighted : bool, optional (default=False)

If True return the sum of edge weights adjacent to the node.

Returns **nd** : dictionary, or number

A dictionary with nodes as keys and out-degree as values or a number if a single node is specified.

Examples

```
>>> G = nx.DiGraph() # or MultiDiGraph
>>> G.add_path([0,1,2,3])
>>> G.out_degree(0)
1
>>> G.out_degree([0,1])
{0: 1, 1: 1}
>>> G.out_degree([0,1]).values()
[1, 1]
```

networkx.MultiDiGraph.out_degree_iter

out_degree_iter (*nbunch=None, weighted=False*)

Return an iterator for (node, out-degree).

The node out-degree is the number of edges pointing out of the node.

Parameters **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

weighted : bool, optional (default=False)

If True return the sum of edge weights adjacent to the node.

Returns **nd_iter** : an iterator

The iterator returns two-tuples of (node, out-degree).

See Also:

`degree`, `in_degree`, `out_degree`, `in_degree_iter`

Examples

```
>>> G = nx.MultiDiGraph()
>>> G.add_path([0,1,2,3])
>>> list(G.out_degree_iter(0)) # node 0 with degree 1
[(0, 1)]
>>> list(G.out_degree_iter([0,1]))
[(0, 1), (1, 1)]
```

networkx.MultiDiGraph.size

size (*weighted=False*)

Return the number of edges.

Parameters `weighted` : boolean, optional (default=False)

If True return the sum of the edge weights.

Returns `nedges` : int

The number of edges in the graph.

See Also:

`number_of_edges`

Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.size()
3
```

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge('a','b',weight=2)
>>> G.add_edge('b','c',weight=4)
>>> G.size()
2
>>> G.size(weighted=True)
6
```

networkx.MultiDiGraph.number_of_edges

number_of_edges (*u=None, v=None*)

Return the number of edges between two nodes.

Parameters `u,v` : nodes, optional (default=all edges)

If u and v are specified, return the number of edges between u and v . Otherwise return the total number of all edges.

Returns `nedges` : int

The number of edges in the graph. If nodes u and v are specified return the number of edges between those nodes.

See Also:

`size`

Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.number_of_edges()
3
>>> G.number_of_edges(0,1)
1
>>> e = (0,1)
>>> G.number_of_edges(*e)
1
```

networkx.MultiDiGraph.nodes_with_selfloops

nodes_with_selfloops ()

Return a list of nodes with self loops.

A node with a self loop has an edge with both ends adjacent to that node.

Returns `nodelist` : list

A list of nodes with self loops.

See Also:

`selfloop_edges`, `number_of_selfloops`

Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge(1,1)
>>> G.add_edge(1,2)
>>> G.nodes_with_selfloops()
[1]
```

networkx.MultiDiGraph.selfloop_edges

selfloop_edges (*data=False*, *keys=False*)

Return a list of selfloop edges.

A selfloop edge has the same node at both ends.

Parameters `data` : bool, optional (default=False)

Return selfloop edges as two tuples (u,v) (`data=False`) or three-tuples (u,v,data) (`data=True`)

keys : bool, optional (default=False)

If True, return edge keys with each edge.

Returns `edgelist` : list of edge tuples

A list of all selfloop edges.

See Also:

`selfloop_nodes`, `number_of_selfloops`

Examples

```
>>> G = nx.MultiGraph()    # or MultiDiGraph
>>> G.add_edge(1,1)
>>> G.add_edge(1,2)
>>> G.selfloop_edges()
[(1, 1)]
>>> G.selfloop_edges(data=True)
[(1, 1, {})]
>>> G.selfloop_edges(keys=True)
[(1, 1, 0)]
>>> G.selfloop_edges(keys=True, data=True)
[(1, 1, 0, {})]
```

`networkx.MultiDiGraph.number_of_selfloops`

`number_of_selfloops()`

Return the number of selfloop edges.

A selfloop edge has the same node at both ends.

Returns `nloops` : int

The number of selfloops.

See Also:

`selfloop_nodes`, `selfloop_edges`

Examples

```
>>> G=nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge(1,1)
>>> G.add_edge(1,2)
>>> G.number_of_selfloops()
1
```

Making copies and subgraphs

<code>MultiDiGraph.copy()</code>	Return a copy of the graph.
<code>MultiDiGraph.to_undirected()</code>	Return an undirected representation of the digraph.
<code>MultiDiGraph.to_directed()</code>	Return a directed copy of the graph.
<code>MultiDiGraph.subgraph(nbunch)</code>	Return the subgraph induced on nodes in nbunch.
<code>MultiDiGraph.reverse([copy])</code>	Return the reverse of the graph.

networkx.MultiDiGraph.copy

`copy()`

Return a copy of the graph.

Returns `G` : Graph

A copy of the graph.

See Also:

`to_directed` return a directed copy of the graph.

Notes

This makes a complete copy of the graph including all of the node or edge attributes.

Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> H = G.copy()
```

networkx.MultiDiGraph.to_undirected

`to_undirected()`

Return an undirected representation of the digraph.

Returns `G` : MultiGraph

An undirected graph with the same name and nodes and with edge (u,v,data) if either (u,v,data) or (v,u,data) is in the digraph. If both edges exist in digraph and their edge data is different, only one edge is created with an arbitrary choice of which edge data to use. You must check and correct for this manually if desired.

Notes

This returns a “deepcopy” of the edge, node, and graph attributes which attempts to completely copy all of the data and references.

This is in contrast to the similar `D=DiGraph(G)` which returns a shallow copy of the data.

See the Python copy module for more information on shallow and deep copies, <http://docs.python.org/library/copy.html>.

networkx.MultiDiGraph.to_directed

to_directed()

Return a directed copy of the graph.

Returns **G** : MultiDiGraph

A deepcopy of the graph.

Notes

If edges in both directions (u,v) and (v,u) exist in the graph, attributes for the new undirected edge will be a combination of the attributes of the directed edges. The edge data is updated in the (arbitrary) order that the edges are encountered. For more customized control of the edge attributes use `add_edge()`.

This returns a “deepcopy” of the edge, node, and graph attributes which attempts to completely copy all of the data and references.

This is in contrast to the similar `G=DiGraph(D)` which returns a shallow copy of the data.

See the Python copy module for more information on shallow and deep copies, <http://docs.python.org/library/copy.html>.

Examples

```
>>> G = nx.Graph() # or MultiGraph, etc
>>> G.add_path([0,1])
>>> H = G.to_directed()
>>> H.edges()
[(0, 1), (1, 0)]
```

If already directed, return a (deep) copy

```
>>> G = nx.MultiDiGraph()
>>> G.add_path([0,1])
>>> H = G.to_directed()
>>> H.edges()
[(0, 1)]
```

networkx.MultiDiGraph.subgraph

subgraph (*nbunch*)

Return the subgraph induced on nodes in nbunch.

The induced subgraph of the graph contains the nodes in nbunch and the edges between those nodes.

Parameters **nbunch** : list, iterable

A container of nodes which will be iterated through once.

Returns **G** : Graph

A subgraph of the graph with the same edge attributes.

Notes

The graph, edge or node attributes just point to the original graph. So changes to the node or edge structure will not be reflected in the original graph while changes to the attributes will.

To create a subgraph with its own copy of the edge/node attributes use: `nx.Graph(G.subgraph(nbunch))`

If edge attributes are containers, a deep copy can be obtained using: `G.subgraph(nbunch).copy()`

For an in-place reduction of a graph to a subgraph you can remove nodes: `G.remove_nodes_from([n in G if n not in set(nbunch)])`

Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> H = G.subgraph([0,1,2])
>>> print H.edges()
[(0, 1), (1, 2)]
```

networkx.MultiDiGraph.reverse

reverse (*copy=True*)

Return the reverse of the graph.

The reverse is a graph with the same nodes and edges but with the directions of the edges reversed.

Parameters **copy** : bool optional (default=True)

If True, return a new DiGraph holding the reversed edges. If False, reverse the reverse graph is created using the original graph (this changes the original graph).

ALGORITHMS

4.1 Bipartite

<code>is_bipartite(G)</code>	Returns True if graph G is bipartite, False if not.
<code>bipartite_sets(G)</code>	Returns bipartite node sets of graph G.
<code>bipartite_color(G)</code>	Returns a two-coloring of the graph.
<code>project(B, nodes[, create_using])</code>	Return the projection of the graph onto a subset of nodes.

4.1.1 networkx.is_bipartite

is_bipartite(G)

Returns True if graph G is bipartite, False if not.

Parameters G : NetworkX graph

See Also:

`bipartite_color`

Examples

```
>>> G=nx.path_graph(4)
>>> print nx.is_bipartite(G)
True
```

4.1.2 networkx.bipartite_sets

bipartite_sets(G)

Returns bipartite node sets of graph G.

Raises an exception if the graph is not bipartite.

Parameters G : NetworkX graph

Returns (X,Y) : two-tuple of sets

One set of nodes for each part of the bipartite graph.

See Also:

`bipartite_color`

Examples

```
>>> G=nx.path_graph(4)
>>> X,Y=nx.bipartite_sets(G)
>>> print X
set([0, 2])
>>> print Y
set([1, 3])
```

4.1.3 networkx.bipartite_color

bipartite_color(*G*)

Returns a two-coloring of the graph.

Raises an exception if the graph is not bipartite.

Parameters *G* : NetworkX graph

Returns *color* : dictionary

A dictionary keyed by node with a 1 or 0 as data for each node color.

Examples

```
>>> G=nx.path_graph(4)
>>> c=nx.bipartite_color(G)
>>> print c
{0: 1, 1: 0, 2: 1, 3: 0}
```

4.1.4 networkx.project

project(*B, nodes, create_using=None*)

Return the projection of the graph onto a subset of nodes.

The nodes retain their names and are connected in the resulting graph if have an edge to a common node in the original graph.

Parameters *B* : NetworkX graph

The input graph should be bipartite.

nodes : list or iterable

Nodes to project onto.

Returns *Graph* : NetworkX graph

A graph that is the projection onto the given nodes.

See Also:

`is_bipartite`, `bipartite_sets`

Notes

Returns a graph that is the projection of the bipartite graph `B` onto the set of nodes given in list `nodes`. No attempt is made to verify that the input graph `B` is bipartite.

Examples

```
>>> B=nx.path_graph(4)
>>> G=nx.project(B, [1, 3])
>>> print G.nodes()
[1, 3]
>>> print G.edges()
[(1, 3)]
```

4.2 Blockmodeling

Functions for creating network blockmodels from node partitions.

Created by Drew Conway <drew.conway@nyu.edu> Copyright (c) 2010. All rights reserved.

<code>blockmodel(G, partitions[, multigraph])</code>	Returns a reduced graph constructed using the generalized block modeling technique.
--	---

4.2.1 networkx.blockmodel

blockmodel (*G, partitions, multigraph=False*)

Returns a reduced graph constructed using the generalized block modeling technique.

The blockmodel technique collapses nodes into blocks based on a given partitioning of the node set. Each partition of nodes (block) is represented as a single node in the reduced graph.

Edges between nodes in the block graph are added according to the edges in the original graph. If the parameter `multigraph` is `False` (the default) a single edge is added with a weight equal to the sum of the edge weights between nodes in the original graph. The default is a weight of 1 if weights are not specified. If the parameter `multigraph` is `True` then multiple edges are added each with the edge data from the original graph.

Parameters **G** : graph

A networkx Graph or DiGraph

partitions : list of lists or list of sets

The partition of the nodes. Must be non-overlapping.

multigraph: bool (optional) :

If `True` return a MultiGraph with the edge data of the original graph applied to each corresponding edge in the new graph. If `False` return a Graph with the sum of the edge weights, or a count of the edges if the original graph is unweighted.

Returns **blockmodel** : a Networkx graph object

References

[R46]

Examples

```
>>> G=nx.path_graph(6)
>>> partition=[[0,1],[2,3],[4,5]]
>>> M=nx.blockmodel(G,partition)
```

4.3 Boundary

Routines to find the boundary of a set of nodes.

Edge boundaries are edges that have only one end in the set of nodes.

Node boundaries are nodes outside the set of nodes that have an edge to a node in the set.

<code>edge_boundary(G, nbunch1[, nbunch2])</code>	Return the edge boundary.
<code>node_boundary(G, nbunch1[, nbunch2])</code>	Return the node boundary.

4.3.1 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.

4.3.2 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.

4.4 Centrality

4.4.1 Degree

<code>degree centrality(G)</code>	Compute the degree centrality for nodes.
<code>in_degree centrality(G)</code>	Compute the in-degree centrality for nodes.
<code>out_degree centrality(G)</code>	Compute the out-degree centrality for nodes.

networkx.degree centrality

degree centrality (*G*)

Compute the degree centrality for nodes.

The degree centrality for a node *v* is the fraction of nodes it is connected to.

Parameters **G** : graph

A networkx graph

Returns **nodes** : dictionary

Dictionary of nodes with degree centrality as the value.

See Also:

`betweenness centrality`, `load centrality`, `eigenvector centrality`

Notes

The degree centrality values are normalized by dividing by the maximum possible degree in a simple graph $n-1$ where n is the number of nodes in G .

For multigraphs or graphs with self loops the maximum degree might be higher than $n-1$ and values of degree centrality greater than 1 are possible.

`networkx.in_degree_centrality`

`in_degree_centrality(G)`

Compute the in-degree centrality for nodes.

The in-degree centrality for a node v is the fraction of nodes its incoming edges are connected to.

Parameters **G** : graph

A NetworkX graph

Returns **nodes** : dictionary

Dictionary of nodes with in-degree centrality as values.

See Also:

`degree_centrality`, `out_degree_centrality`, `Notes`, `-----`, `The`, `possible`, `For`, `be`, `are`

`networkx.out_degree_centrality`

`out_degree_centrality(G)`

Compute the out-degree centrality for nodes.

The out-degree centrality for a node v is the fraction of nodes its outgoing edges are connected to.

Parameters **G** : graph

A NetworkX graph

Returns **nodes** : dictionary

Dictionary of nodes with out-degree centrality as values.

See Also:

`degree_centrality`, `in_degree_centrality`

Notes

The degree centrality values are normalized by dividing by the maximum possible degree in a simple graph $n-1$ where n is the number of nodes in G .

For multigraphs or graphs with self loops the maximum degree might be higher than $n-1$ and values of degree centrality greater than 1 are possible.

4.4.2 Closeness

Closeness centrality measures.

`closeness_centrality(G[, v, weighted_edges, ...])` Compute closeness centrality for nodes.

networkx.closeness centrality**closeness centrality** (*G*, *v=None*, *weighted_edges=False*, *normalized=True*)

Compute closeness centrality for nodes.

Closeness centrality at a node is 1/average distance to all other nodes.

Parameters **G** : graph

A networkx graph

v : node, optional

Return only the value for node v.

weighted_edges : bool, optional

Consider the edge weights in determining the shortest paths. If False, all edge weights are considered equal.

normalized : bool, optional

If True normalize the values to the size of the connected component containing v.

Returns **nodes** : dictionary

Dictionary of nodes with closeness centrality as the value.

See Also:[betweenness centrality](#),
[degree centrality](#)[load centrality](#),[eigenvector centrality](#),**Notes**

The closeness centrality is normalized to $n-1 / \text{size}(G)-1$ where n is the number of nodes in the connected part of graph containing the node. If the graph is not completely connected, this algorithm computes the closeness centrality for each connected part separately.

4.4.3 Betweenness

Betweenness centrality measures.

betweenness centrality (<i>G</i> [, <i>normalized</i> , ...])	Compute betweenness centrality for nodes.
edge_betweenness centrality (<i>G</i> [, <i>normalized</i> , ...])	Compute betweenness centrality for edges.

networkx.algorithms centrality.betweenness.betweenness centrality**betweenness centrality** (*G*, *normalized=True*, *weighted_edges=False*, *endpoints=False*)

Compute betweenness centrality for nodes.

Betweenness centrality of a node is the fraction of all shortest paths that pass through that node.

Parameters **G** : graph

A networkx graph

normalized : bool, optionalIf True the betweenness values are normalized by $b=b/(n-1)(n-2)$ where n is the number of nodes in G .

weighted_edges : bool, optional

Consider the edge weights in determining the shortest paths. The edge weights must be greater than zero. If False, all edge weights are considered equal.

Returns nodes : dictionary

Dictionary of nodes with betweenness centrality as the value.

See Also:

`edge_betweenness_centrality`, `load_centrality`

Notes

The algorithm is from Ulrik Brandes [R35].

For weighted graphs the edge weights must be greater than zero. Zero edge weights can produce an infinite number of equal length paths between pairs of nodes.

References

[R35]

`networkx.algorithms.centrality.betweenness.edge_betweenness_centrality`

edge_betweenness_centrality (*G*, *normalized=True*, *weighted_edges=False*)

Compute betweenness centrality for edges.

Betweenness centrality of an edge is the fraction of all shortest paths that pass through that edge.

Parameters G : graph

A networkx graph

normalized : bool, optional

If True the betweenness values are normalized by $b=b/(n-1)(n-2)$ where n is the number of nodes in G .

weighted_edges : bool, optional

Consider the edge weights in determining the shortest paths. The edge weights must be greater than zero. If False, all edge weights are considered equal.

Returns edges : dictionary

Dictionary of edges with betweenness centrality as the value.

See Also:

`betweenness_centrality`, `edge_load`

Notes

The algorithm is from Ulrik Brandes [R36].

For weighted graphs the edge weights must be greater than zero. Zero edge weights can produce an infinite number of equal length paths between pairs of nodes.

References

[R36]

4.4.4 Current Flow Closeness

Current-flow closeness centrality measures.

`current_flow_closeness_centrality(G[, ...])` Compute current-flow closeness centrality for nodes.

`networkx.current_flow_closeness_centrality`

`current_flow_closeness_centrality` (*G*, *normalized=True*)

Compute current-flow closeness centrality for nodes.

A variant of closeness centrality based on effective resistance between nodes in a network. This metric is also known as information centrality.

Parameters **G** : graph

A networkx graph

normalized : bool, optional

If True the values are normalized by $1/(n-1)$ where n is the number of nodes in G .

Returns **nodes** : dictionary

Dictionary of nodes with current flow closeness centrality as the value.

See Also:

`closeness_centrality`

Notes

The algorithm is from Brandes [R47].

If the edges have a ‘weight’ attribute they will be used as weights in this algorithm. Unspecified weights are set to 1.

See also [R48] for the original definition of information centrality.

References

[R47], [R48]

4.4.5 Current-Flow Betweenness

Current-flow betweenness centrality measures.

`current_flow_betweenness_centrality(G[, ...])` Compute current-flow betweenness centrality for nodes.

`edge_current_flow_betweenness_centrality(G)` Compute current-flow betweenness centrality for edges.

`networkx.algorithms centrality.current_flow_betweenness.current_flow_betweenness centrality`

`current_flow_betweenness centrality` (*G*, *normalized=True*)

Compute current-flow betweenness centrality for nodes.

Current-flow betweenness centrality uses an electrical current model for information spreading in contrast to betweenness centrality which uses shortest paths.

Current-flow betweenness centrality is also known as random-walk betweenness centrality [R38].

Parameters *G* : graph

A networkx graph

normalized : bool, optional

If True the betweenness values are normalized by $b=b/(n-1)(n-2)$ where *n* is the number of nodes in *G*.

Returns *nodes* : dictionary

Dictionary of nodes with betweenness centrality as the value.

See Also:

`betweenness centrality`, `edge_betweenness centrality`, `edge_current_flow_betweenness centra`

Notes

The algorithm is from Brandes [R37].

If the edges have a ‘weight’ attribute they will be used as weights in this algorithm. Unspecified weights are set to 1.

References

[R37], [R38]

`networkx.algorithms centrality.current_flow_betweenness.edge_current_flow_betweenness centrality`

`edge_current_flow_betweenness centrality` (*G*, *normalized=True*)

Compute current-flow betweenness centrality for edges.

Current-flow betweenness centrality uses an electrical current model for information spreading in contrast to betweenness centrality which uses shortest paths.

Current-flow betweenness centrality is also known as random-walk betweenness centrality [R40].

Parameters *G* : graph

A networkx graph

normalized : bool, optional

If True the betweenness values are normalized by $b=b/(n-1)(n-2)$ where *n* is the number of nodes in *G*.

Returns *nodes* : dictionary

Dictionary of edge tuples with betweenness centrality as the value.

See Also:

`betweenness centrality`, `edge_betweenness centrality`, `current_flow_betweenness centrality`

Notes

The algorithm is from Brandes [R39].

If the edges have a ‘weight’ attribute they will be used as weights in this algorithm. Unspecified weights are set to 1.

References

[R39], [R40]

4.4.6 Eigenvector

Eigenvector centrality.

<code>eigenvector centrality(G[, max_iter, tol, ...])</code>	Compute the eigenvector centrality for the graph G.
<code>eigenvector centrality_numpy(G)</code>	Compute the eigenvector centrality for the graph G.

networkx.eigenvector centrality

eigenvector centrality (*G*, *max_iter=100*, *tol=9.999999999999995e-07*, *nstart=None*)

Compute the eigenvector centrality for the graph G.

Uses the power method to find the eigenvector for the largest eigenvalue of the adjacency matrix of G.

Parameters **G** : graph

A networkx graph

max_iter : interger, optional

Maximum number of iterations in power method.

tol : float, optional

Error tolerance used to check convergence in power method iteration.

nstart : dictionary, optional

Starting value of eigenvector iteration for each node.

Returns **nodes** : dictionary

Dictionary of nodes with eigenvector centrality as the value.

See Also:

`eigenvector centrality_numpy`, `pagerank`, `hits`

Notes

The eigenvector calculation is done by the power iteration method and has no guarantee of convergence. The iteration will stop after `max_iter` iterations or an error tolerance of `number_of_nodes(G)*tol` has been reached.

For directed graphs this is “right” eigenvector centrality. For “left” eigenvector centrality, first reverse the graph with `G.reverse()`.

Examples

```
>>> G=nx.path_graph(4)
>>> centrality=nx.eigenvector_centrality(G)
>>> print(['%s %0.2f'%(node,centrality[node]) for node in centrality])
['0 0.37', '1 0.60', '2 0.60', '3 0.37']
```

`networkx.eigenvector_centrality_numpy`

`eigenvector_centrality_numpy(G)`

Compute the eigenvector centrality for the graph `G`.

Parameters `G` : graph

A networkx graph

Returns `nodes` : dictionary

Dictionary of nodes with eigenvector centrality as the value.

See Also:

`eigenvector_centrality`, `pagerank`, `hits`

Notes

This algorithm uses the NumPy eigenvalue solver.

For directed graphs this is “right” eigenvector centrality. For “left” eigenvector centrality, first reverse the graph with `G.reverse()`.

Examples

```
>>> G=nx.path_graph(4)
>>> centrality=nx.eigenvector_centrality_numpy(G)
>>> print(['%s %0.2f'%(node,centrality[node]) for node in centrality])
['0 0.37', '1 0.60', '2 0.60', '3 0.37']
```

4.4.7 Load

Load centrality.

<code>load_centrality(G[, v, cutoff, normalized, ...])</code>	Compute load centrality for nodes.
<code>edge_load(G[, nodes, cutoff])</code>	Compute edge load.

networkx.algorithms centrality.load.load centrality**load centrality** (*G*, *v=None*, *cutoff=None*, *normalized=True*, *weighted_edges=False*)

Compute load centrality for nodes.

The load centrality of a node is the fraction of all shortest paths that pass through that node.

Parameters **G** : graph

A networkx graph

normalized : bool, optionalIf True the betweenness values are normalized by $b=b/(n-1)(n-2)$ where *n* is the number of nodes in *G*.**weighted_edges** : bool, optional

Consider the edge weights in determining the shortest paths. If False, all edge weights are considered equal.

cutoff : bool, optionalIf specified, only consider paths of length \leq cutoff.**Returns** **nodes** : dictionary

Dictionary of nodes with centrality as the value.

See Also:`betweenness centrality`**Notes**

Load centrality is slightly different than betweenness. For this load algorithm see the reference Scientific collaboration networks: II. Shortest paths, weighted networks, and centrality, M. E. J. Newman, Phys. Rev. E 64, 016132 (2001).

networkx.algorithms centrality.load.edge_load**edge_load** (*G*, *nodes=None*, *cutoff=False*)

Compute edge load.

WARNING:

This module is for demonstration and testing purposes.

4.5 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

<code>find_cliques(G)</code>	Search for all maximal cliques in a graph.
<code>make_max_clique_graph(G, create_using, name)</code>	Create the maximal clique graph of a graph.
<code>make_clique_bipartite(G, fpos, ...)</code>	Create a bipartite clique graph from a graph G.
<code>graph_clique_number(G, cliques)</code>	Return the clique number (size of the largest clique) for G.
<code>graph_number_of_cliques(G, cliques)</code>	Returns the number of maximal cliques in G.
<code>node_clique_number(G, nodes, cliques)</code>	Returns the size of the largest maximal clique containing each given node.
<code>number_of_cliques(G, nodes, cliques)</code>	Returns the number of maximal cliques for each node.
<code>cliques_containing_node(G, nodes, cliques)</code>	Returns a list of cliques containing the given node.

4.5.1 networkx.find_cliques

find_cliques (G)

Search for all maximal cliques in a graph.

This algorithm searches for maximal cliques in a graph. maximal cliques are the largest complete subgraph containing a given point. The largest maximal clique is sometimes called the maximum clique.

This implementation is a generator of lists each of which contains the members of a maximal clique. To obtain a list of cliques, use `list(find_cliques(G))`. The method essentially unrolls the recursion used in the references to avoid issues of recursion stack depth.

See Also:

`find_cliques_recursive, A`

References

Based on the algorithm published by Bron & Kerbosch (1973) <http://doi.acm.org/10.1145/362342.362367>

as adapted by Tomita, Tanaka and Takahashi (2006) <http://dx.doi.org/10.1016/j.tcs.2006.06.015>

and discussed in Cazals and Karande (2008) <http://dx.doi.org/10.1016/j.tcs.2008.05.010>

4.5.2 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.

4.5.3 networkx.make_clique_bipartite

make_clique_bipartite (G, fpos=None, create_using=None, name=None)

Create a bipartite clique graph from a graph G.

Nodes of G are retained as the “bottom nodes” of B and cliques of G become “top nodes” of B . Edges are present if a bottom node belongs to the clique represented by the top node.

Returns a Graph with additional attribute dict $B.node_type$ which is keyed by nodes to “Bottom” or “Top” appropriately.

if $fpos$ is not None, a second additional attribute dict $B.pos$ is created to hold the position tuple of each node for viewing the bipartite graph.

4.5.4 networkx.graph_clique_number

graph_clique_number (G , *cliques=None*)

Return the clique number (size of the largest clique) for G .

An optional list of cliques can be input if already computed.

4.5.5 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.

4.5.6 networkx.node_clique_number

node_clique_number (G , *nodes=None*, *cliques=None*)

Returns the size of the largest maximal clique containing each given node.

Returns a single or list depending on input nodes. Optional list of cliques can be input if already computed.

4.5.7 networkx.number_of_cliques

number_of_cliques (G , *nodes=None*, *cliques=None*)

Returns the number of maximal cliques for each node.

Returns a single or list depending on input nodes. Optional list of cliques can be input if already computed.

4.5.8 networkx.cliques_containing_node

cliques_containing_node (G , *nodes=None*, *cliques=None*)

Returns a list of cliques containing the given node.

Returns a single list or list of lists depending on input nodes. Optional list of cliques can be input if already computed.

4.6 Clustering

Algorithms to characterize the number of triangles in a graph.

<code>triangles(G[, nbunch])</code>	Compute the number of triangles.
<code>transitivity(G)</code>	Compute transitivity.
<code>clustering(G[, nbunch, weights])</code>	Compute the clustering coefficient for nodes.
<code>average_clustering(G)</code>	Compute average clustering coefficient.

4.6.1 networkx.triangles

triangles (*G*, *nbunch=None*)

Compute the number of triangles.

Finds the number of triangles that include a node as one of the vertices.

Parameters **G** : graph

A networkx graph

nbunch : container of nodes, optional

Compute triangles for nodes in nbunch. The default is all nodes in G.

Returns out : dictionary

Number of triangles keyed by node label.

Notes

When computing triangles for the entire graph each triangle is counted three times, once at each node.

Self loops are ignored.

Examples

```
>>> G=nx.complete_graph(5)
>>> print nx.triangles(G,0)
6
>>> print nx.triangles(G)
{0: 6, 1: 6, 2: 6, 3: 6, 4: 6}
>>> print nx.triangles(G, (0,1)).values()
[6, 6]
```

4.6.2 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).

$T = 3 * \text{triangles} / \text{triads}$

Parameters **G** : graph

A networkx graph

Returns out : float

Transitivity

Examples

```
>>> G=nx.complete_graph(5)
>>> print nx.transitivity(G)
1.0
```

4.6.3 networkx.clustering

clustering (*G*, *nbunch=None*, *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.

weights : bool, optional

If True return fraction of connected triples as dictionary

Returns **out** : float, 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. This is useful for computing transitivity.

Self loops are ignored.

Examples

```
>>> G=nx.complete_graph(5)
>>> print nx.clustering(G,0)
1.0
>>> print nx.clustering(G)
{0: 1.0, 1: 1.0, 2: 1.0, 3: 1.0, 4: 1.0}
```

4.6.4 networkx.average_clustering

average_clustering (*G*)

Compute average clustering coefficient.

A clustering coefficient for the whole graph is the average,

$$C = \frac{1}{n} \sum_{v \in G} c_v,$$

where n is the number of nodes in G .

Parameters G : graph

A networkx graph

Returns **out** : float

Average clustering

Notes

This is a space saving routine; it might be faster to use clustering to get a list and then take the average.

Self loops are ignored.

Examples

```
>>> G=nx.complete_graph(5)
>>> print nx.average_clustering(G)
1.0
```

4.7 Components

4.7.1 Connectivity

Connected components.

<code>is_connected(G)</code>	Test graph connectivity
<code>number_connected_components(G)</code>	Return number of connected components in graph.
<code>connected_components(G)</code>	Return nodes in connected components of graph.
<code>connected_component_subgraphs(G)</code>	Return connected components as subgraphs.
<code>node_connected_component(G, n)</code>	Return nodes in connected components of graph containing node n.

`networkx.algorithms.components.connected.is_connected`

`is_connected(G)`

Test graph connectivity

Parameters G : NetworkX Graph

An undirected graph.

Returns **connected** : bool

True if the graph is connected, false otherwise.

See Also:

`connected_components`

Notes

For undirected graphs only.

Examples

```
>>> G=nx.path_graph(4)
>>> print nx.is_connected(G)
True
```

networkx.algorithms.components.connected.number_connected_components

number_connected_components (*G*)

Return number of connected components in graph.

Parameters **G** : NetworkX Graph

An undirected graph.

Returns **n** : integer

Number of connected components

See Also:

[connected_components](#)

Notes

For undirected graphs only.

networkx.algorithms.components.connected.connected_components

connected_components (*G*)

Return nodes in connected components of graph.

Parameters **G** : NetworkX Graph

An undirected graph.

Returns **comp** : list of lists

A list of nodes for each component of G.

See Also:

[strongly_connected_components](#)

Notes

The list is ordered from largest connected component to smallest. For undirected graphs only.

networkx.algorithms.components.connected.connected_component_subgraphs

connected_component_subgraphs (*G*)

Return connected components as subgraphs.

Parameters **G** : NetworkX Graph

An undirected graph.

Returns **glist** : list

A list of graphs, one for each connected component of G.

See Also:

`connected_components`

Notes

The list is ordered from largest connected component to smallest. For undirected graphs only.

Examples

Get largest connected component as subgraph

```
>>> G=nx.path_graph(4)
>>> G.add_edge(5,6)
>>> H=nx.connected_component_subgraphs(G)[0]
```

networkx.algorithms.components.connected.node_connected_component

node_connected_component (*G*, *n*)

Return nodes in connected components of graph containing node n.

Parameters **G** : NetworkX Graph

An undirected graph.

n : node label

A node in G

Returns **comp** : lists

A list of nodes in component of G containing node n.

See Also:

`connected_components`

Notes

For undirected graphs only.

4.7.2 Strong connectivity

Strongly connected components.

<code>is_strongly_connected(G)</code>	Test directed graph for strong connectivity.
<code>number_strongly_connected_components(G)</code>	Return number of strongly connected components in graph.
<code>strongly_connected_components(G)</code>	Return nodes in strongly connected components of graph.
<code>strongly_connected_component_subgraphs(G)</code>	Return strongly connected components as subgraphs.
<code>strongly_connected_components_recursive(G)</code>	Return nodes in strongly connected components of graph.
<code>kosaraju_strongly_connected_components(G[, ...])</code>	Return nodes in strongly connected components of graph.
<code>condensation(G)</code>	Returns the condensation of G.

`networkx.algorithms.components.strongly_connected.is_strongly_connected`

`is_strongly_connected` (*G*)

Test directed graph for strong connectivity.

Parameters *G* : NetworkX Graph

A directed graph.

Returns `connected` : bool

True if the graph is strongly connected, False otherwise.

See Also:

`strongly_connected_components`

Notes

For directed graphs only.

`networkx.algorithms.components.strongly_connected.number_strongly_connected_components`

`number_strongly_connected_components` (*G*)

Return number of strongly connected components in graph.

Parameters *G* : NetworkX graph

A directed graph.

Returns *n* : integer

Number of strongly connected components

See Also:

`connected_components`

Notes

For directed graphs only.

networkx.algorithms.components.strongly_connected.strongly_connected_components

strongly_connected_components (*G*)

Return nodes in strongly connected components of graph.

Parameters **G** : NetworkX Graph

An directed graph.

Returns **comp** : list of lists

A list of nodes for each component of *G*. The list is ordered from largest connected component to smallest.

See Also:

`connected_components`

Notes

Uses Tarjan's algorithm with Nuutila's modifications. Nonrecursive version of algorithm.

References

[R41], [R42]

networkx.algorithms.components.strongly_connected.strongly_connected_component_subgraphs

strongly_connected_component_subgraphs (*G*)

Return strongly connected components as subgraphs.

Parameters **G** : NetworkX Graph

A graph.

Returns **glist** : list

A list of graphs, one for each strongly connected component of *G*.

See Also:

`connected_component_subgraphs`

Notes

The list is ordered from largest strongly connected component to smallest.

networkx.algorithms.components.strongly_connected.strongly_connected_components_recursive

strongly_connected_components_recursive (*G*)

Return nodes in strongly connected components of graph.

Recursive version of algorithm.

Parameters **G** : NetworkX Graph

An directed graph.

Returns comp : list of lists

A list of nodes for each component of G. The list is ordered from largest connected component to smallest.

See Also:

`connected_components`

Notes

Uses Tarjan's algorithm with Nuutila's modifications.

References

[R43], [R44]

`networkx.algorithms.components.strongly_connected.kosaraju_strongly_connected_components`

kosaraju_strongly_connected_components (*G*, *source=None*)

Return nodes in strongly connected components of graph.

Parameters G : NetworkX Graph

An directed graph.

Returns comp : list of lists

A list of nodes for each component of G. The list is ordered from largest connected component to smallest.

See Also:

`connected_components`

Notes

Uses Kosaraju's algorithm.

`networkx.algorithms.components.strongly_connected.condensation`

condensation (*G*)

Returns the condensation of G.

The condensation of G is the graph with each of the strongly connected components contracted into a single node.

Parameters G : NetworkX Graph

A directed graph.

Returns cG : NetworkX DiGraph

The condensation of G.

Notes

After contracting all strongly connected components to a single node, the resulting graph is a directed acyclic graph.

4.7.3 Weak connectivity

Weakly connected components.

<code>is_weakly_connected(G)</code>	Test directed graph for weak connectivity.
<code>number_weakly_connected_components(G)</code>	Return the number of connected components in G.
<code>weakly_connected_components(G)</code>	Return weakly connected components of G.
<code>weakly_connected_component_subgraphs(G)</code>	Return weakly connected components as subgraphs.

`networkx.algorithms.components.weakly_connected.is_weakly_connected`

`is_weakly_connected(G)`

Test directed graph for weak connectivity.

Parameters `G` : NetworkX Graph

A directed graph.

Returns `connected` : bool

True if the graph is weakly connected, False otherwise.

See Also:

`strongly_connected_components`

Notes

For directed graphs only.

`networkx.algorithms.components.weakly_connected.number_weakly_connected_components`

`number_weakly_connected_components(G)`

Return the number of connected components in G. For directed graphs only.

`networkx.algorithms.components.weakly_connected.weakly_connected_components`

`weakly_connected_components(G)`

Return weakly connected components of G.

`networkx.algorithms.components.weakly_connected.weakly_connected_component_subgraphs`

`weakly_connected_component_subgraphs(G)`

Return weakly connected components as subgraphs.

4.7.4 Attracting components

Attracting components.

<code>is_attracting_component(G)</code>	Returns True if G consists of a single attracting component.
<code>number_attracting_components(G)</code>	Returns the number of attracting components in G .
<code>attracting_components(G)</code>	Returns a list of attracting components in G .
<code>attracting_component_subgraphs(G)</code>	Returns a list of attracting component subgraphs from G .

`networkx.algorithms.components.attracting.is_attracting_component`

`is_attracting_component` (G)

Returns True if G consists of a single attracting component.

Parameters G : DiGraph, MultiDiGraph

The graph to be analyzed.

Returns `attracting` : bool

True if G has a single attracting component. Otherwise, False.

See Also:

`attracting_components`, `number_attracting_components`, `attracting_component_subgraphs`

`networkx.algorithms.components.attracting.number_attracting_components`

`number_attracting_components` (G)

Returns the number of attracting components in G .

Parameters G : DiGraph, MultiDiGraph

The graph to be analyzed.

Returns n : int

The number of attracting components in G .

See Also:

`attracting_components`, `is_attracting_component`, `attracting_component_subgraphs`

`networkx.algorithms.components.attracting.attracting_components`

`attracting_components` (G)

Returns a list of attracting components in G .

An attracting component in a directed graph G is a strongly connected component with the property that a random walker on the graph will never leave the component, once it enters the component.

The nodes in attracting components can also be thought of as recurrent nodes. If a random walker enters the attractor containing the node, then the node will be visited infinitely often.

Parameters G : DiGraph, MultiDiGraph

The graph to be analyzed.

Returns `attractors` : list

The list of attracting components, sorted from largest attracting component to smallest attracting component.

See Also:

`number_attracting_components,` `is_attracting_component,`
`attracting_component_subgraphs`

networkx.algorithms.components.attracting.attracting_component_subgraphs

attracting_component_subgraphs (*G*)

Returns a list of attracting component subgraphs from *G*.

Parameters *G* : DiGraph, MultiDiGraph

The graph to be analyzed.

Returns *subgraphs* : list

A list of node-induced subgraphs of the attracting components of *G*.

See Also:

`attracting_components,` `number_attracting_components,` `is_attracting_component`

4.8 Cores

Find the *k*-cores of a graph. The *k*-core is found by recursively pruning nodes with degrees less than *k*.

`find_cores(G)` Return the core number for each vertex.

4.8.1 networkx.find_cores

find_cores (*G*)

Return the core number for each vertex.

Parameters *G* : NetworkX graph

A graph

Returns *core_number* : dictionary

A dictionary keyed by node to the core number.

References

[R53]

4.9 Cycles

`cycle_basis(G[, root])` Returns a list of cycles which form a basis for cycles of *G*.

4.9.1 networkx.cycle_basis

cycle_basis (*G*, *root=None*)

Returns a list of cycles which form a basis for cycles of *G*.

A basis for cycles of a network is a minimal collection of cycles such that any cycle in the network can be written as a sum of cycles in the basis. Here summation of cycles is defined as “exclusive or” of the edges. Cycle bases are useful, e.g. when deriving equations for electric circuits using Kirchhoff’s Laws.

Parameters *G* : NetworkX Graph

root : node of *G*, optional (default=arbitrary choice from *G*)

Returns A list of cycle lists. Each cycle list is a list of nodes :

which forms a cycle (loop) in *G* . :

Notes

This algorithm is adapted from algorithm CACM 491 published: Paton, K. An algorithm for finding a fundamental set of cycles of a graph. Comm. ACM 12, 9 (Sept 1969), 514-518.

Examples

```
>>> G=nx.Graph()
>>> G.add_cycle([0,1,2,3])
>>> G.add_cycle([0,3,4,5])
>>> print nx.cycle_basis(G,0)
[[3, 4, 5, 0], [1, 2, 3, 0]]
```

4.10 Directed Acyclic Graphs

Algorithms for directed acyclic graphs (DAGs).

<code>topological_sort(G[, nbunch])</code>	Return a list of nodes in topological sort order.
<code>topological_sort_recursive(G[, nbunch])</code>	Return a list of nodes in topological sort order.
<code>is_directed_acyclic_graph(G)</code>	Return True if the graph <i>G</i> is a directed acyclic graph (DAG)

4.10.1 networkx.topological_sort

topological_sort (*G*, *nbunch=None*)

Return a list of nodes in topological sort order.

A topological sort is a nonunique permutation of the nodes such that an edge from *u* to *v* implies that *u* appears before *v* in the topological sort order.

Parameters *G* : NetworkX digraph

A directed graph

nbunch : container of nodes (optional)

Explore graph in specified order given in *nbunch*

See Also:

`is_directed_acyclic_graph`

Notes

If G is not a directed acyclic graph (DAG) no topological sort exists and the Python keyword `None` is returned. This algorithm is based on a description and proof in The Algorithm Design Manual [R95] .

References

[R95]

4.10.2 `networkx.topological_sort_recursive`

`topological_sort_recursive` (G , *nbunch=None*)

Return a list of nodes in topological sort order.

A topological sort is a nonunique permutation of the nodes such that an edge from u to v implies that u appears before v in the topological sort order.

Parameters G : NetworkX digraph

nbunch : container of nodes (optional)

Explore graph in specified order given in `nbunch`

See Also:

`topological_sort`, `is_directed_acyclic_graph`

Notes

If G is not a directed acyclic graph (DAG) no topological sort exists and the Python keyword `None` is returned. This is a recursive version of topological sort.

4.10.3 `networkx.is_directed_acyclic_graph`

`is_directed_acyclic_graph` (G)

Return True if the graph G is a directed acyclic graph (DAG) or False if not.

Parameters G : NetworkX graph

A graph

Returns `is_dag` : bool

True if G is a DAG, false otherwise

4.11 Distance Measures

Graph diameter, radius, eccentricity and other properties.

<code>center(G[, e])</code>	Return the periphery of the graph G.
<code>diameter(G[, e])</code>	Return the diameter of the graph G.
<code>eccentricity(G[, v, sp])</code>	Return the eccentricity of nodes in G.
<code>periphery(G[, e])</code>	Return the periphery of the graph G.
<code>radius(G[, e])</code>	Return the radius of the graph G.

4.11.1 networkx.center

center (*G*, *e=None*)

Return the periphery of the graph G.

The center is the set of nodes with eccentricity equal to radius.

Parameters **G** : NetworkX graph

A graph

e : eccentricity dictionary, optional

A precomputed dictionary of eccentricities.

Returns **c** : list

List of nodes in center

4.11.2 networkx.diameter

diameter (*G*, *e=None*)

Return the diameter of the graph G.

The diameter is the maximum eccentricity.

Parameters **G** : NetworkX graph

A graph

e : eccentricity dictionary, optional

A precomputed dictionary of eccentricities.

Returns **d** : integer

Diameter of graph

See Also:

`eccentricity`

4.11.3 networkx.eccentricity

eccentricity (*G*, *v=None*, *sp=None*)

Return the eccentricity of nodes in G.

The eccentricity of a node *v* is the maximum distance from *v* to all other nodes in G.

Parameters **G** : NetworkX graph

A graph

v : node, optional

Return value of specified node

sp : dict of dicts, optional

All pairs shortest path lengths as a dictionary of dictionaries

Returns ecc : dictionary

A dictionary of eccentricity values keyed by node.

4.11.4 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.

Parameters G : NetworkX graph

A graph

e : eccentricity dictionary, optional

A precomputed dictionary of eccentricities.

Returns p : list

List of nodes in periphery

4.11.5 networkx.radius

radius (*G*, *e=None*)

Return the radius of the graph *G*.

The radius is the minimum eccentricity.

Parameters G : NetworkX graph

A graph

e : eccentricity dictionary, optional

A precomputed dictionary of eccentricities.

Returns r : integer

Radius of graph

4.12 Eulerian

Eulerian circuits and graphs.

<code>is_eulerian(G)</code>	Return True if <i>G</i> is an Eulerian graph, False otherwise.
<code>eulerian_circuit(G[, source])</code>	Return the edges of an Eulerian circuit in <i>G</i> .

4.12.1 networkx.is_eulerian

is_eulerian (*G*)

Return True if *G* is an Eulerian graph, False otherwise.

An Eulerian graph is a graph with an Eulerian circuit.

Parameters *G* : NetworkX graph

Notes

This implementation requires the graph to be connected (or strongly connected for directed graphs).

Examples

```
>>> is_eulerian(nx.DiGraph({0:[3], 1:[2], 2:[3], 3:[0, 1]}))
True
>>> is_eulerian(nx.complete_graph(5))
True
>>> is_eulerian(nx.petersen_graph())
False
```

4.12.2 networkx.eulerian_circuit

eulerian_circuit (*G*, *source=None*)

Return the edges of an Eulerian circuit in *G*.

An Eulerian circuit is a path that crosses every edge in *G* exactly once and finishes at the starting node.

Parameters *G* : NetworkX graph

source : node, optional

Starting node for circuit.

Returns *edges* : generator

A generator that produces edges in the Eulerian circuit.

Notes

Uses Fleury's algorithm [R51],[R52]

References

[R51], [R52]

Examples

```
>>> G=nx.complete_graph(3)
>>> list( eulerian_circuit(G) )
[(0, 1), (1, 2), (2, 0)]
>>> list( eulerian_circuit(G,source=1) )
[(1, 0), (0, 2), (2, 1)]
>>> [u for u,v in eulerian_circuit(G)] # nodes in circuit
[0, 1, 2]
```

4.13 Flows

4.13.1 Ford-Fulkerson

<code>max_flow(G, s, t)</code>	Find a maximum single-commodity flow using the Ford-Fulkerson
<code>min_cut(G, s, t)</code>	Compute the value of a minimum (s, t)-cut.
<code>ford_fulkerson(G, s, t)</code>	Find a maximum single-commodity flow using the Ford-Fulkerson

networkx.max_flow

max_flow (*G, s, t*)

Find a maximum single-commodity flow using the Ford-Fulkerson algorithm.

This algorithm uses Edmond-Karp-Dinitz path selection rule which guarantees a running time of $O(|V||E|^2)$.

Parameters **G** : NetworkX graph

Edges of the graph are expected to have an attribute called 'capacity'. If this attribute is not present, the edge is considered to have infinite capacity.

s : node

Source node for the flow.

t : node

Sink node for the flow.

Returns **flowValue** : integer, float

Value of the maximum flow, i.e., net outflow from the source.

flowGraph : NetworkX graph

Graph with $V(\text{flowGraph}) = V(G)$ and in which each edge has an attribute 'flow' which gives the flow on the edge.

Raises **NetworkXError** :

If the graph has a path of infinite capacity, the value of a feasible flow on the graph is unbounded above and the function raises a NetworkXError.

Examples

```
>>> import networkx as nx
>>> G = nx.DiGraph()
>>> G.add_edge('x', 'a', capacity = 3.0)
```

```

>>> G.add_edge('x','b', capacity = 1.0)
>>> G.add_edge('a','c', capacity = 3.0)
>>> G.add_edge('b','c', capacity = 5.0)
>>> G.add_edge('b','d', capacity = 4.0)
>>> G.add_edge('d','e', capacity = 2.0)
>>> G.add_edge('c','y', capacity = 2.0)
>>> G.add_edge('e','y', capacity = 3.0)
>>> flow,F=nx.ford_ulkerson(G, 'x', 'y')
>>> flow
3.0

```

networkx.min_cut

min_cut (*G*, *s*, *t*)

Compute the value of a minimum (*s*, *t*)-cut.

Use the max-flow min-cut theorem, i.e., the capacity of a minimum capacity cut is equal to the flow value of a maximum flow.

Parameters **G** : NetworkX graph

Edges of the graph are expected to have an attribute called 'capacity'. If this attribute is not present, the edge is considered to have infinite capacity.

s : node

Source node for the flow.

t : node

Sink node for the flow.

Returns **cutValue** : integer, float

Value of the minimum cut.

Raises **NetworkXError** :

If the graph has a path of infinite capacity, all cuts have infinite capacity and the function raises a NetworkXError.

Examples

```

>>> import networkx as nx
>>> G = nx.DiGraph()
>>> G.add_edge('x','a', capacity = 3.0)
>>> G.add_edge('x','b', capacity = 1.0)
>>> G.add_edge('a','c', capacity = 3.0)
>>> G.add_edge('b','c', capacity = 5.0)
>>> G.add_edge('b','d', capacity = 4.0)
>>> G.add_edge('d','e', capacity = 2.0)
>>> G.add_edge('c','y', capacity = 2.0)
>>> G.add_edge('e','y', capacity = 3.0)
>>> nx.min_cut(G, 'x', 'y')
3.0

```

networkx.ford_fulkerson

ford_fulkerson (*G*, *s*, *t*)

Find a maximum single-commodity flow using the Ford-Fulkerson algorithm.

This algorithm uses Edmond-Karp-Dinitz path selection rule which guarantees a running time of $O(|V||E|^2)$.

Parameters **G** : NetworkX graph

Edges of the graph are expected to have an attribute called 'capacity'. If this attribute is not present, the edge is considered to have infinite capacity.

s : node

Source node for the flow.

t : node

Sink node for the flow.

Returns **flowValue** : integer, float

Value of the maximum flow, i.e., net outflow from the source.

flowGraph : NetworkX graph

Graph with $V(\text{flowGraph}) = V(G)$ and in which each edge has an attribute 'flow' which gives the flow on the edge.

Raises **NetworkXError** :

If the graph has a path of infinite capacity, the value of a feasible flow on the graph is unbounded above and the function raises a NetworkXError.

Examples

```
>>> import networkx as nx
>>> G = nx.DiGraph()
>>> G.add_edge('x', 'a', capacity = 3.0)
>>> G.add_edge('x', 'b', capacity = 1.0)
>>> G.add_edge('a', 'c', capacity = 3.0)
>>> G.add_edge('b', 'c', capacity = 5.0)
>>> G.add_edge('b', 'd', capacity = 4.0)
>>> G.add_edge('d', 'e', capacity = 2.0)
>>> G.add_edge('c', 'y', capacity = 2.0)
>>> G.add_edge('e', 'y', capacity = 3.0)
>>> flow, F=nx.ford_fulkerson(G, 'x', 'y')
>>> flow
3.0
```

4.14 Isolates

Functions for identifying isolate (degree zero) nodes.

<code>is_isolate(G, n)</code>	Determine if node <i>n</i> is an isolate (degree zero).
<code>isolates(G)</code>	Return list of isolates in the graph.

4.14.1 networkx.is_isolate

is_isolate(*G*, *n*)

Determine if node *n* is an isolate (degree zero).

Parameters **G** : graph

A networkx graph

n : node

A node in G

Returns **isolate** : bool

True if *n* has no neighbors, False otherwise.

Examples

```
>>> G=nx.Graph()
>>> G.add_edge(1,2)
>>> G.add_node(3)
>>> nx.is_isolate(G,2)
False
>>> nx.is_isolate(G,3)
True
```

4.14.2 networkx.isolates

isolates(*G*)

Return list of isolates in the graph.

Isolates are nodes with no neighbors (degree zero).

Parameters **G** : graph

A networkx graph

Returns **isolates** : list

List of isolate nodes.

Examples

```
>>> G=nx.Graph()
>>> G.add_edge(1,2)
>>> G.add_node(3)
>>> nx.isolates(G)
[3]
```

To remove all isolates in the graph use >>> G.remove_nodes_from(nx.isolates(G)) >>> G.nodes() [1, 2]

4.15 Isomorphism

<code>is_isomorphic(G1, G2[, weighted, rtol, atol])</code>	Returns True if the graphs G1 and G2 are isomorphic and False otherwise.
<code>could_be_isomorphic(G1, G2)</code>	Returns False if graphs are definitely not isomorphic.
<code>fast_could_be_isomorphic(G1, G2)</code>	Returns False if graphs are definitely not isomorphic.
<code>faster_could_be_isomorphic(G1, G2)</code>	Returns False if graphs are definitely not isomorphic.

4.15.1 networkx.is_isomorphic

`is_isomorphic(G1, G2, weighted=False, rtol=9.999999999999995e-07, atol=1.0000000000000001e-09)`

Returns True if the graphs G1 and G2 are isomorphic and False otherwise.

Parameters **G1, G2:** NetworkX graph instances :

The two graphs G1 and G2 must be the same type.

weighted: bool, optional :

Optionally check isomorphism for weighted graphs. G1 and G2 must be valid weighted graphs.

rtol: float, optional :

The relative error tolerance when checking weighted edges

atol: float, optional :

The absolute error tolerance when checking weighted edges

See Also:

`isomorphvf2`

Notes

Uses the vf2 algorithm. Works for Graph, DiGraph, MultiGraph, and MultiDiGraph

4.15.2 networkx.could_be_isomorphic

`could_be_isomorphic(G1, G2)`

Returns False if graphs are definitely not isomorphic. True does NOT guarantee isomorphism.

Parameters **G1, G2 :** NetworkX graph instances

The two graphs G1 and G2 must be the same type.

Notes

Checks for matching degree, triangle, and number of cliques sequences.

4.15.3 networkx.fast_could_be_isomorphic

fast_could_be_isomorphic(G1, G2)

Returns False if graphs are definitely not isomorphic. True does NOT guarantee isomorphism.

Parameters G1, G2 : NetworkX graph instances

The two graphs G1 and G2 must be the same type.

Notes

Checks for matching degree and triangle sequences.

4.15.4 networkx.faster_could_be_isomorphic

faster_could_be_isomorphic(G1, G2)

Returns False if graphs are definitely not isomorphic. True does NOT guarantee isomorphism.

Parameters G1, G2 : NetworkX graph instances

The two graphs G1 and G2 must be the same type.

Notes

Checks for matching degree sequences.

4.15.5 Advanced Interface to VF2 Algorithm

VF2 Algorithm

Graph Matcher

<code>GraphMatcher.__init__(G1, G2)</code>	Initialize GraphMatcher.
<code>GraphMatcher.initialize()</code>	Reinitializes the state of the algorithm.
<code>GraphMatcher.is_isomorphic()</code>	Returns True if G1 and G2 are isomorphic graphs.
<code>GraphMatcher.subgraph_is_isomorphic()</code>	Returns True if a subgraph of G1 is isomorphic to G2.
<code>GraphMatcher.isomorphisms_iter()</code>	Generator over isomorphisms between G1 and G2.
<code>GraphMatcher.subgraph_isomorphisms_iter()</code>	Generator over isomorphisms between a subgraph of G1 and G2.
<code>GraphMatcher.candidate_pairs_iter()</code>	Iterator over candidate pairs of nodes in G1 and G2.
<code>GraphMatcher.match()</code>	Extends the isomorphism mapping.
<code>GraphMatcher.semantic_feasibility(G1_node, G2_node, ...)</code>	Returns True if adding (G1_node, G2_node) is syntactically feasible.
<code>GraphMatcher.syntactic_feasibility(G1_node, G2_node, ...)</code>	Returns True if adding (G1_node, G2_node) is syntactically feasible.

networkx.GraphMatcher.__init__

__init__(G1, G2)

Initialize GraphMatcher.

Parameters G1,G2: NetworkX Graph or MultiGraph instances. :

The two graphs to check for isomorphism.

Examples

To create a GraphMatcher which checks for syntactic feasibility:

```
>>> G1 = nx.path_graph(4)
>>> G2 = nx.path_graph(4)
>>> GM = nx.GraphMatcher(G1, G2)
```

networkx.GraphMatcher.initialize

initialize()

Reinitializes the state of the algorithm.

This method should be redefined if using something other than GMState. If only subclassing GraphMatcher, a redefinition is not necessary.

networkx.GraphMatcher.is_isomorphic

is_isomorphic()

Returns True if G1 and G2 are isomorphic graphs.

networkx.GraphMatcher.subgraph_is_isomorphic

subgraph_is_isomorphic()

Returns True if a subgraph of G1 is isomorphic to G2.

networkx.GraphMatcher.isomorphisms_iter

isomorphisms_iter()

Generator over isomorphisms between G1 and G2.

networkx.GraphMatcher.subgraph_isomorphisms_iter

subgraph_isomorphisms_iter()

Generator over isomorphisms between a subgraph of G1 and G2.

networkx.GraphMatcher.candidate_pairs_iter

candidate_pairs_iter()

Iterator over candidate pairs of nodes in G1 and G2.

networkx.GraphMatcher.match

match()

Extends the isomorphism mapping.

This function is called recursively to determine if a complete isomorphism can be found between G1 and G2. It cleans up the class variables after each recursive call. If an isomorphism is found, we yield the mapping.

networkx.GraphMatcher.semantic_feasibility**semantic_feasibility** (*G1_node, G2_node*)Returns True if adding (*G1_node, G2_node*) is symantically feasible.

The semantic feasibility function should return True if it is acceptable to add the candidate pair (*G1_node, G2_node*) to the current partial isomorphism mapping. The logic should focus on semantic information contained in the edge data or a formalized node class.

By acceptable, we mean that the subsequent mapping can still become a complete isomorphism mapping. Thus, if adding the candidate pair definitely makes it so that the subsequent mapping cannot become a complete isomorphism mapping, then this function must return False.

The default semantic feasibility function always returns True. The effect is that semantics are not considered in the matching of *G1* and *G2*.

The semantic checks might differ based on the what type of test is being performed. A keyword description of the test is stored in `self.test`. Here is a quick description of the currently implemented tests:

test='graph' Indicates that the graph matcher is looking for a graph-graph isomorphism.

test='subgraph' Indicates that the graph matcher is looking for a subgraph-graph isomorphism such that a subgraph of *G1* is isomorphic to *G2*.

Any subclass which redefines `semantic_feasibility()` must maintain the above form to keep the `match()` method functional. Implementations should consider multigraphs.

networkx.GraphMatcher.syntactic_feasibility**syntactic_feasibility** (*G1_node, G2_node*)Returns True if adding (*G1_node, G2_node*) is syntactically feasible.

This function returns True if it is adding the candidate pair to the current partial isomorphism mapping is allowable. The addition is allowable if the inclusion of the candidate pair does not make it impossible for an isomorphism to be found.

DiGraph Matcher

<code>DiGraphMatcher.__init__(G1, G2)</code>	Initialize DiGraphMatcher.
<code>DiGraphMatcher.initialize()</code>	Reinitializes the state of the algorithm.
<code>DiGraphMatcher.is_isomorphic()</code>	Returns True if <i>G1</i> and <i>G2</i> are isomorphic graphs.
<code>DiGraphMatcher.subgraph_is_isomorphic()</code>	Returns True if a subgraph of <i>G1</i> is isomorphic to <i>G2</i> .
<code>DiGraphMatcher.isomorphisms_iter()</code>	Generator over isomorphisms between <i>G1</i> and <i>G2</i> .
<code>DiGraphMatcher.subgraph_isomorphisms_iter()</code>	Generator over isomorphisms between a subgraph of <i>G1</i> and <i>G2</i> .
<code>DiGraphMatcher.candidate_pairs_iter()</code>	Iterator over candidate pairs of nodes in <i>G1</i> and <i>G2</i> .
<code>DiGraphMatcher.match()</code>	Extends the isomorphism mapping.
<code>DiGraphMatcher.semantic_feasibility(G1_node, G2_node, ...)</code>	Returns True if adding (<i>G1_node, G2_node</i>) is symantically feasible.
<code>DiGraphMatcher.syntactic_feasibility(...)</code>	Returns True if adding (<i>G1_node, G2_node</i>) is syntactically feasible.

networkx.DiGraphMatcher.__init__**__init__** (*G1, G2*)

Initialize DiGraphMatcher.

G1 and *G2* should be `nx.Graph` or `nx.MultiGraph` instances.

Examples

To create a GraphMatcher which checks for syntactic feasibility:

```
>>> G1 = nx.DiGraph(nx.path_graph(4, create_using=nx.DiGraph()))
>>> G2 = nx.DiGraph(nx.path_graph(4, create_using=nx.DiGraph()))
>>> DiGM = nx.DiGraphMatcher(G1, G2)
```

networkx.DiGraphMatcher.initialize

initialize()

Reinitializes the state of the algorithm.

This method should be redefined if using something other than DiGMState. If only subclassing GraphMatcher, a redefinition is not necessary.

networkx.DiGraphMatcher.is_isomorphic

is_isomorphic()

Returns True if G1 and G2 are isomorphic graphs.

networkx.DiGraphMatcher.subgraph_is_isomorphic

subgraph_is_isomorphic()

Returns True if a subgraph of G1 is isomorphic to G2.

networkx.DiGraphMatcher.isomorphisms_iter

isomorphisms_iter()

Generator over isomorphisms between G1 and G2.

networkx.DiGraphMatcher.subgraph_isomorphisms_iter

subgraph_isomorphisms_iter()

Generator over isomorphisms between a subgraph of G1 and G2.

networkx.DiGraphMatcher.candidate_pairs_iter

candidate_pairs_iter()

Iterator over candidate pairs of nodes in G1 and G2.

networkx.DiGraphMatcher.match

match()

Extends the isomorphism mapping.

This function is called recursively to determine if a complete isomorphism can be found between G1 and G2. It cleans up the class variables after each recursive call. If an isomorphism is found, we yield the mapping.

networkx.DiGraphMatcher.semantic_feasibility

semantic_feasibility(G1_node, G2_node)

Returns True if adding (G1_node, G2_node) is syntactically feasible.

The semantic feasibility function should return True if it is acceptable to add the candidate pair (G1_node, G2_node) to the current partial isomorphism mapping. The logic should focus on semantic information contained in the edge data or a formalized node class.

By acceptable, we mean that the subsequent mapping can still become a complete isomorphism mapping. Thus, if adding the candidate pair definitely makes it so that the subsequent mapping cannot become a complete isomorphism mapping, then this function must return False.

The default semantic feasibility function always returns True. The effect is that semantics are not considered in the matching of G1 and G2.

The semantic checks might differ based on the what type of test is being performed. A keyword description of the test is stored in self.test. Here is a quick description of the currently implemented tests:

test='graph' Indicates that the graph matcher is looking for a graph-graph isomorphism.

test='subgraph' Indicates that the graph matcher is looking for a subgraph-graph isomorphism such that a subgraph of G1 is isomorphic to G2.

Any subclass which redefines semantic_feasibility() must maintain the above form to keep the match() method functional. Implementations should consider multigraphs.

networkx.DiGraphMatcher.syntactic_feasibility

syntactic_feasibility (*G1_node*, *G2_node*)

Returns True if adding (*G1_node*, *G2_node*) is syntactically feasible.

This function returns True if it is adding the candidate pair to the current partial isomorphism mapping is allowable. The addition is allowable if the inclusion of the candidate pair does not make it impossible for an isomorphism to be found.

Weighted Graph Matcher

<code>WeightedGraphMatcher.__init__(G1, G2[, ...])</code>	Initialize WeightedGraphMatcher.
<code>WeightedGraphMatcher.initialize()</code>	Reinitializes the state of the algorithm.
<code>WeightedGraphMatcher.is_isomorphic()</code>	Returns True if G1 and G2 are isomorphic graphs.
<code>WeightedGraphMatcher.subgraph_is_isomorphic()</code>	Returns True if a subgraph of G1 is isomorphic to G2.
<code>WeightedGraphMatcher.isomorphisms_iter()</code>	Generator over isomorphisms between G1 and G2.
<code>WeightedGraphMatcher.subgraph_isomorphisms_iter()</code>	Generator over isomorphisms between a subgraph of G1 and G2.
<code>WeightedGraphMatcher.candidate_pairs_iter()</code>	Iterator over candidate pairs of nodes in G1 and G2.
<code>WeightedGraphMatcher.match()</code>	Extends the isomorphism mapping.
<code>WeightedGraphMatcher.semantic_feasibility(G1_node, G2_node)</code>	Returns True if mapping G1_node to G2_node is semantically feasible.
<code>WeightedGraphMatcher.syntactic_feasibility(G1_node, G2_node)</code>	Returns True if adding (G1_node, G2_node) is syntactically feasible.

networkx.WeightedGraphMatcher.__init__

__init__ (*G1*, *G2*, *rtol*=9.9999999999999995e-07, *atol*=1.0000000000000001e-09)

Initialize WeightedGraphMatcher.

Parameters **G1**, **G2** : nx.Graph instances

G1 and G2 must be weighted graphs.

rtol : float, optional

The relative tolerance used to compare weights.

atol : float, optional

The absolute tolerance used to compare weights.

networkx.WeightedGraphMatcher.initialize

initialize ()

Reinitializes the state of the algorithm.

This method should be redefined if using something other than GMState. If only subclassing GraphMatcher, a redefinition is not necessary.

networkx.WeightedGraphMatcher.is_isomorphic

is_isomorphic ()

Returns True if G1 and G2 are isomorphic graphs.

networkx.WeightedGraphMatcher.subgraph_is_isomorphic

subgraph_is_isomorphic ()

Returns True if a subgraph of G1 is isomorphic to G2.

networkx.WeightedGraphMatcher.isomorphisms_iter

isomorphisms_iter ()

Generator over isomorphisms between G1 and G2.

networkx.WeightedGraphMatcher.subgraph_isomorphisms_iter

subgraph_isomorphisms_iter ()

Generator over isomorphisms between a subgraph of G1 and G2.

networkx.WeightedGraphMatcher.candidate_pairs_iter

candidate_pairs_iter ()

Iterator over candidate pairs of nodes in G1 and G2.

networkx.WeightedGraphMatcher.match

match ()

Extends the isomorphism mapping.

This function is called recursively to determine if a complete isomorphism can be found between G1 and G2. It cleans up the class variables after each recursive call. If an isomorphism is found, we yield the mapping.

networkx.WeightedGraphMatcher.semantic_feasibility

semantic_feasibility (G1_node, G2_node)

Returns True if mapping G1_node to G2_node is semantically feasible.

networkx.WeightedGraphMatcher.syntactic_feasibility

syntactic_feasibility (G1_node, G2_node)

Returns True if adding (G1_node, G2_node) is syntactically feasible.

This function returns True if it is adding the candidate pair to the current partial isomorphism mapping is allowable. The addition is allowable if the inclusion of the candidate pair does not make it impossible for an isomorphism to be found.

Weighted DiGraph Matcher

<code>WeightedDiGraphMatcher.__init__(G1, G2[, ...])</code>	Initialize WeightedGraphMatcher.
<code>WeightedDiGraphMatcher.initialize()</code>	Reinitializes the state of the algorithm.
<code>WeightedDiGraphMatcher.is_isomorphic()</code>	Returns True if G1 and G2 are isomorphic graphs.
<code>WeightedDiGraphMatcher.subgraph_is_isomorphic()</code>	Returns True if a subgraph of G1 is isomorphic to G2.
<code>WeightedDiGraphMatcher.isomorphisms_iterator()</code>	Generator over isomorphisms between G1 and G2.
<code>WeightedDiGraphMatcher.subgraph_isomorphisms_iterator()</code>	Generator over isomorphisms between a subgraph of G1 and G2.
<code>WeightedDiGraphMatcher.candidate_pairs_iterator()</code>	Iterator over candidate pairs of nodes in G1 and G2.
<code>WeightedDiGraphMatcher.match()</code>	Extends the isomorphism mapping.
<code>WeightedDiGraphMatcher.semantic_feasibility(G1_node, G2_node)</code>	Returns True if mapping G1_node to G2_node is semantically feasible.
<code>WeightedDiGraphMatcher.syntactic_feasibility(G1_node, G2_node)</code>	Returns True if adding (G1_node, G2_node) is syntactically feasible.

`networkx.WeightedDiGraphMatcher.__init__`

`__init__(G1, G2, rtol=9.999999999999995e-07, atol=1.0000000000000001e-09)`
Initialize WeightedGraphMatcher.

Parameters **G1, G2** : nx.DiGraph instances

G1 and G2 must be weighted graphs.

rtol : float, optional

The relative tolerance used to compare weights.

atol : float, optional

The absolute tolerance used to compare weights.

`networkx.WeightedDiGraphMatcher.initialize`

`initialize()`

Reinitializes the state of the algorithm.

This method should be redefined if using something other than DiGMState. If only subclassing GraphMatcher, a redefinition is not necessary.

`networkx.WeightedDiGraphMatcher.is_isomorphic`

`is_isomorphic()`

Returns True if G1 and G2 are isomorphic graphs.

`networkx.WeightedDiGraphMatcher.subgraph_is_isomorphic`

`subgraph_is_isomorphic()`

Returns True if a subgraph of G1 is isomorphic to G2.

`networkx.WeightedDiGraphMatcher.isomorphisms_iterator`

`isomorphisms_iterator()`

Generator over isomorphisms between G1 and G2.

networkx.WeightedDiGraphMatcher.subgraph_isomorphisms_iter
subgraph_isomorphisms_iter ()

Generator over isomorphisms between a subgraph of G1 and G2.

networkx.WeightedDiGraphMatcher.candidate_pairs_iter
candidate_pairs_iter ()

Iterator over candidate pairs of nodes in G1 and G2.

networkx.WeightedDiGraphMatcher.match
match ()

Extends the isomorphism mapping.

This function is called recursively to determine if a complete isomorphism can be found between G1 and G2. It cleans up the class variables after each recursive call. If an isomorphism is found, we yield the mapping.

networkx.WeightedDiGraphMatcher.semantic_feasibility
semantic_feasibility (G1_node, G2_node)

Returns True if mapping G1_node to G2_node is semantically feasible.

networkx.WeightedDiGraphMatcher.syntactic_feasibility
syntactic_feasibility (G1_node, G2_node)

Returns True if adding (G1_node, G2_node) is syntactically feasible.

This function returns True if it is adding the candidate pair to the current partial isomorphism mapping is allowable. The addition is allowable if the inclusion of the candidate pair does not make it impossible for an isomorphism to be found.

Weighted MultiGraph Matcher

<code>WeightedMultiGraphMatcher.__init__(G1, G2[, ...])</code>	Initialize WeightedGraphMatcher.
<code>WeightedMultiGraphMatcher.initialize()</code>	Reinitializes the state of the algorithm.
<code>WeightedMultiGraphMatcher.is_isomorphic()</code>	Returns True if G1 and G2 are isomorphic graphs.
<code>WeightedMultiGraphMatcher.subgraph_is_isomorphic()</code>	Returns True if a subgraph of G1 is isomorphic to G2.
<code>WeightedMultiGraphMatcher.isomorphisms_iter(G)</code>	Generator over isomorphisms between G1 and G2.
<code>WeightedMultiGraphMatcher.subgraph_isomorphisms_iter()</code>	Generator over isomorphisms between a subgraph of G1 and G2.
<code>WeightedMultiGraphMatcher.candidate_pairs_iter()</code>	Iterator over candidate pairs of nodes in G1 and G2.
<code>WeightedMultiGraphMatcher.match()</code>	Extends the isomorphism mapping.
<code>WeightedMultiGraphMatcher.semantic_feasibility(...)</code>	
<code>WeightedMultiGraphMatcher.syntactic_feasibility(...)</code>	Returns True if adding (G1_node, G2_node) is syntactically feasible.

networkx.WeightedMultiGraphMatcher.__init__
__init__ (G1, G2, rtol=9.999999999999995e-07, atol=1.0000000000000001e-09)
 Initialize WeightedGraphMatcher.

Parameters G1, G2 : nx.MultiGraph instances

G1 and G2 must be weighted graphs.

rtol : float, optional

The relative tolerance used to compare weights.

atol : float, optional

The absolute tolerance used to compare weights.

networkx.WeightedMultiGraphMatcher.initialize

initialize ()

Reinitializes the state of the algorithm.

This method should be redefined if using something other than GMState. If only subclassing GraphMatcher, a redefinition is not necessary.

networkx.WeightedMultiGraphMatcher.is_isomorphic

is_isomorphic ()

Returns True if G1 and G2 are isomorphic graphs.

networkx.WeightedMultiGraphMatcher.subgraph_is_isomorphic

subgraph_is_isomorphic ()

Returns True if a subgraph of G1 is isomorphic to G2.

networkx.WeightedMultiGraphMatcher.isomorphisms_iter

isomorphisms_iter ()

Generator over isomorphisms between G1 and G2.

networkx.WeightedMultiGraphMatcher.subgraph_isomorphisms_iter

subgraph_isomorphisms_iter ()

Generator over isomorphisms between a subgraph of G1 and G2.

networkx.WeightedMultiGraphMatcher.candidate_pairs_iter

candidate_pairs_iter ()

Iterator over candidate pairs of nodes in G1 and G2.

networkx.WeightedMultiGraphMatcher.match

match ()

Extends the isomorphism mapping.

This function is called recursively to determine if a complete isomorphism can be found between G1 and G2. It cleans up the class variables after each recursive call. If an isomorphism is found, we yield the mapping.

networkx.WeightedMultiGraphMatcher.semantic_feasibility

semantic_feasibility (*G1_node*, *G2_node*)

networkx.WeightedMultiGraphMatcher.syntactic_feasibility

syntactic_feasibility (*G1_node*, *G2_node*)

Returns True if adding (*G1_node*, *G2_node*) is syntactically feasible.

This function returns True if it is adding the candidate pair to the current partial isomorphism mapping is allowable. The addition is allowable if the inclusion of the candidate pair does not make it impossible for an isomorphism to be found.

Weighted MultiDiGraph Matcher

<code>WeightedMultiDiGraphMatcher.__init__(G1, G2)</code>	Initialize WeightedGraphMatcher.
<code>WeightedMultiDiGraphMatcher.initialize()</code>	Reinitializes the state of the algorithm.
<code>WeightedMultiDiGraphMatcher.is_isomorphic()</code>	Returns True if G1 and G2 are isomorphic graphs.
<code>WeightedMultiDiGraphMatcher.subgraph_is_isomorphic()</code>	Returns True if a subgraph of G1 is isomorphic to G2.
<code>WeightedMultiDiGraphMatcher.isomorphisms_generator()</code>	Generator over isomorphisms between G1 and G2.
<code>WeightedMultiDiGraphMatcher.subgraph_isomorphisms_generator()</code>	Generator over isomorphisms between a subgraph of G1 and G2.
<code>WeightedMultiDiGraphMatcher.candidate_pairs_iterator()</code>	Iterator over candidate pairs of nodes in G1 and G2.
<code>WeightedMultiDiGraphMatcher.match()</code>	Extends the isomorphism mapping.
<code>WeightedMultiDiGraphMatcher.semantic_feasible()</code>	Returns True if mapping G1_node to G2_node is semantically feasible.
<code>WeightedMultiDiGraphMatcher.syntactic_feasible()</code>	Returns True if adding (G1_node, G2_node) is syntactically feasible.

networkx.WeightedMultiDiGraphMatcher.__init__

`__init__(G1, G2, rtol=9.999999999999995e-07, atol=1.0000000000000001e-09)`

Initialize WeightedGraphMatcher.

Parameters **G1, G2** : nx.MultiDiGraph instances

G1 and G2 must be weighted graphs.

rtol : float, optional

The relative tolerance used to compare weights.

atol : float, optional

The absolute tolerance used to compare weights.

networkx.WeightedMultiDiGraphMatcher.initialize

`initialize()`

Reinitializes the state of the algorithm.

This method should be redefined if using something other than DiGMState. If only subclassing GraphMatcher, a redefinition is not necessary.

networkx.WeightedMultiDiGraphMatcher.is_isomorphic

`is_isomorphic()`

Returns True if G1 and G2 are isomorphic graphs.

networkx.WeightedMultiDiGraphMatcher.subgraph_is_isomorphic

`subgraph_is_isomorphic()`

Returns True if a subgraph of G1 is isomorphic to G2.

networkx.WeightedMultiDiGraphMatcher.isomorphisms_iter
isomorphisms_iter ()

Generator over isomorphisms between G1 and G2.

networkx.WeightedMultiDiGraphMatcher.subgraph_isomorphisms_iter
subgraph_isomorphisms_iter ()

Generator over isomorphisms between a subgraph of G1 and G2.

networkx.WeightedMultiDiGraphMatcher.candidate_pairs_iter
candidate_pairs_iter ()

Iterator over candidate pairs of nodes in G1 and G2.

networkx.WeightedMultiDiGraphMatcher.match
match ()

Extends the isomorphism mapping.

This function is called recursively to determine if a complete isomorphism can be found between G1 and G2. It cleans up the class variables after each recursive call. If an isomorphism is found, we yield the mapping.

networkx.WeightedMultiDiGraphMatcher.semantic_feasibility
semantic_feasibility (*G1_node*, *G2_node*)

Returns True if mapping *G1_node* to *G2_node* is semantically feasible.

networkx.WeightedMultiDiGraphMatcher.syntactic_feasibility
syntactic_feasibility (*G1_node*, *G2_node*)

Returns True if adding (*G1_node*, *G2_node*) is syntactically feasible.

This function returns True if it is adding the candidate pair to the current partial isomorphism mapping is allowable. The addition is allowable if the inclusion of the candidate pair does not make it impossible for an isomorphism to be found.

4.16 Link Analysis

4.16.1 PageRank

PageRank analysis of graph structure.

<code>pagerank(G[, alpha, max_iter, tol, nstart])</code>	Return the PageRank of the nodes in the graph.
<code>pagerank_numpy(G[, alpha])</code>	Return the PageRank of the nodes in the graph.
<code>pagerank_scipy(G[, alpha, max_iter, tol, ...])</code>	Return the PageRank of the nodes in the graph.
<code>google_matrix(G[, alpha, nodelist])</code>	Return the Google matrix of the graph.

networkx.pagerank

pagerank (*G*, *alpha*=0.8499999999999998, *max_iter*=100, *tol*=1e-08, *nstart*=None)

Return the PageRank of the nodes in the graph.

PageRank computes a ranking of the nodes in the graph *G* based on the structure of the incoming links. It was originally designed as an algorithm to rank web pages.

Parameters **G** : graph

A NetworkX graph

alpha : float, optional

Damping parameter for PageRank, default=0.85

max_iter : integer, optional

Maximum number of iterations in power method eigenvalue solver.

tol : float, optional

Error tolerance used to check convergence in power method solver.

nstart : dictionary, optional

Starting value of PageRank iteration for each node.

Returns **nodes** : dictionary

Dictionary of nodes with value as PageRank

Notes

The eigenvector calculation is done by the power iteration method and has no guarantee of convergence. The iteration will stop after `max_iter` iterations or an error tolerance of `number_of_nodes(G)*tol` has been reached.

The PageRank algorithm was designed for directed graphs but this algorithm does not check if the input graph is directed and will execute on undirected graphs by converting each oriented edge in the directed graph to two edges.

References

[R89], [R90]

Examples

```
>>> G=nx.DiGraph(nx.path_graph(4))
>>> pr=nx.pagerank(G, alpha=0.9)
```

`networkx.pagerank_numpy`

`pagerank_numpy(G, alpha=0.8499999999999998)`

Return the PageRank of the nodes in the graph.

PageRank computes a ranking of the nodes in the graph `G` based on the structure of the incoming links. It was originally designed as an algorithm to rank web pages.

Parameters **G** : graph

A NetworkX graph

alpha : float, optional

Damping parameter for PageRank, default=0.85

Returns nodes : dictionary

Dictionary of nodes with value as PageRank

Notes

The eigenvector calculation uses NumPy's interface to the LAPACK eigenvalue solvers.

This implementation works with Multi(Di)Graphs.

References

[R91], [R92]

Examples

```
>>> G=nx.DiGraph(nx.path_graph(4))
>>> pr=nx.pagerank_numpy(G, alpha=0.9)
```

networkx.pagerank_scipy

pagerank_scipy (*G*, *alpha*=0.8499999999999998, *max_iter*=100, *tol*=9.999999999999995e-07, *nodelist*=None)

Return the PageRank of the nodes in the graph.

PageRank computes a ranking of the nodes in the graph *G* based on the structure of the incoming links. It was originally designed as an algorithm to rank web pages.

Parameters *G* : graph

A NetworkX graph

alpha : float, optional

Damping parameter for PageRank, default=0.85

Returns nodes : dictionary

Dictionary of nodes with value as PageRank

Notes

The eigenvector calculation uses power iteration with a SciPy sparse matrix representation.

References

[R93], [R94]

Examples

```
>>> G=nx.DiGraph(nx.path_graph(4))
>>> pr=nx.pagerank_numpy(G,alpha=0.9)
```

networkx.google_matrix

google_matrix (*G*, *alpha*=0.8499999999999998, *nodelist*=None)

Return the Google matrix of the graph.

Parameters **G** : graph

A NetworkX graph

alpha : float

The damping factor

nodelist : list, optional

The rows and columns are ordered according to the nodes in nodelist. If nodelist is None, then the ordering is produced by G.nodes().

Returns **A** : NumPy matrix

Google matrix of the graph

4.16.2 Hits

Hubs and authorities analysis of graph structure.

<code>hits(G[, max_iter, tol, nstart])</code>	Return HITS hubs and authorities values for nodes.
<code>hits_numpy(G)</code>	Return HITS hubs and authorities values for nodes.
<code>hits_scipy(G[, max_iter, tol])</code>	Return HITS hubs and authorities values for nodes.
<code>hub_matrix(G[, nodelist])</code>	Return the HITS hub matrix.
<code>authority_matrix(G[, nodelist])</code>	Return the HITS authority matrix.

networkx.hits

hits (*G*, *max_iter*=100, *tol*=1e-08, *nstart*=None)

Return HITS hubs and authorities values for nodes.

The HITS algorithm computes two numbers for a node. Authorities estimates the node value based on the incoming links. Hubs estimates the node value based on outgoing links.

Parameters **G** : graph

A NetworkX graph

max_iter : interger, optional

Maximum number of iterations in power method.

tol : float, optional

Error tolerance used to check convergence in power method iteration.

nstart : dictionary, optional

Starting value of each node for power method iteration.

Returns (hubs,authorities) : two-tuple of dictionaries

Two dictionaries keyed by node containing the hub and authority values.

Notes

The eigenvector calculation is done by the power iteration method and has no guarantee of convergence. The iteration will stop after `max_iter` iterations or an error tolerance of `number_of_nodes(G)*tol` has been reached.

The HITS algorithm was designed for directed graphs but this algorithm does not check if the input graph is directed and will execute on undirected graphs.

References

[R80], [R81]

Examples

```
>>> G=nx.path_graph(4)
>>> h,a=nx.hits(G)
```

networkx.hits_numpy

hits_numpy(*G*)

Return HITS hubs and authorities values for nodes.

The HITS algorithm computes two numbers for a node. Authorities estimates the node value based on the incoming links. Hubs estimates the node value based on outgoing links.

Parameters **G** : graph

A NetworkX graph

Returns (hubs,authorities) : two-tuple of dictionaries

Two dictionaries keyed by node containing the hub and authority values.

Notes

The eigenvector calculation uses NumPy's interface to LAPACK.

The HITS algorithm was designed for directed graphs but this algorithm does not check if the input graph is directed and will execute on undirected graphs.

References

[R82], [R83]

Examples

```
>>> G=nx.path_graph(4)
>>> h,a=nx.hits(G)
```

networkx.hits_scipy

hits_scipy (*G*, *max_iter*=100, *tol*=9.999999999999995e-07)

Return HITS hubs and authorities values for nodes.

The HITS algorithm computes two numbers for a node. Authorities estimates the node value based on the incoming links. Hubs estimates the node value based on outgoing links.

Parameters **G** : graph

A NetworkX graph

max_iter : interger, optional

Maximum number of iterations in power method.

tol : float, optional

Error tolerance used to check convergence in power method iteration.

nstart : dictionary, optional

Starting value of each node for power method iteration.

Returns (**hubs,authorities**) : two-tuple of dictionaries

Two dictionaries keyed by node containing the hub and authority values.

Notes

This implementation uses SciPy sparse matrices.

The eigenvector calculation is done by the power iteration method and has no guarantee of convergence. The iteration will stop after *max_iter* iterations or an error tolerance of $\text{number_of_nodes}(G)*\text{tol}$ has been reached.

The HITS algorithm was designed for directed graphs but this algorithm does not check if the input graph is directed and will execute on undirected graphs.

References

[R84], [R85]

Examples

```
>>> G=nx.path_graph(4)
>>> h,a=nx.hits(G)
```

networkx.hub_matrix

hub_matrix (*G*, *nodelist=None*)
Return the HITS hub matrix.

networkx.authority_matrix

authority_matrix (*G*, *nodelist=None*)
Return the HITS authority matrix.

4.17 Matching

The algorithm is taken from “Efficient Algorithms for Finding Maximum Matching in Graphs” by Zvi Galil, ACM Computing Surveys, 1986. It is based on the “blossom” method for finding augmenting paths and the “primal-dual” method for finding a matching of maximum weight, both methods invented by Jack Edmonds.

`max_weight_matching(G[, maxcardinality])` Compute a maximum-weighted matching of G.

4.17.1 networkx.max_weight_matching

max_weight_matching (*G*, *maxcardinality=False*)
Compute a maximum-weighted matching of G.

A matching is a subset of edges in which no node occurs more than once. The cardinality of a matching is the number of matched edges. The weight of a matching is the sum of the weights of its edges.

Parameters **G** : NetworkX graph

Undirected graph

maxcardinality: bool, optional :

If *maxcardinality* is True, compute the maximum-cardinality matching with maximum weight among all maximum-cardinality matchings.

Returns **mate** : dictionary

The matching is returned as a dictionary, *mate*, such that *mate*[*v*] == *w* if node *v* is matched to node *w*. Unmatched nodes do not occur as a key in *mate*.

Notes

If *G* has edges with ‘weight’ attribute the edge data are used as weight values else the weights are assumed to be 1.

This function takes time $O(\text{number_of_nodes}^{**} 3)$.

If all edge weights are integers, the algorithm uses only integer computations. If floating point weights are used, the algorithm could return a slightly suboptimal matching due to numeric precision errors.

References

[R87]

4.18 Mixing Patterns

Mixing matrices and assortativity coefficients.

4.18.1 Assortativity

<code>degree_assortativity(G)</code>	Compute degree assortativity of graph.
<code>attribute_assortativity(G, attribute)</code>	Compute assortativity for node attributes.
<code>numeric_assortativity(G, attribute)</code>	Compute assortativity for numerical node attributes.
<code>neighbor_connectivity(G)</code>	Compute neighbor connectivity of graph.
<code>degree_pearsonr(G)</code>	Compute degree assortativity of graph.

`networkx.degree_assortativity`

`degree_assortativity(G)`

Compute degree assortativity of graph.

Assortativity measures the similarity of connections in the graph with respect to the node degree.

Parameters `G` : NetworkX graph

Returns `r` : float

Assortativity of graph by degree.

See Also:

`attribute_assortativity`, `numeric_assortativity`, `neighbor_connectivity`,
`degree_mixing_dict`, `degree_mixing_matrix`

Notes

This computes Eq. (21) in Ref. [R49], where e is the joint probability distribution (mixing matrix) of the degrees. If G is directed then the matrix e is the joint probability of out-degree and in-degree.

References

[R49]

Examples

```
>>> G=nx.path_graph(4)
>>> r=nx.degree_assortativity(G)
>>> print "%3.1f"%r
-0.5
```


networkx.attribute_assortativity**attribute_assortativity**(*G*, *attribute*)

Compute assortativity for node attributes.

Assortativity measures the similarity of connections in the graph with respect to the given attribute.

Parameters **G** : NetworkX graph**attribute** : string

Node attribute key

Returns **a**: float :

Assortativity of given attribute

Notes

This computes Eq. (2) in Ref. [R45], $(\text{trace}(e) - \sum(e)) / (1 - \sum(e))$, where e is the joint probability distribution (mixing matrix) of the specified attribute.

References

[R45]

Examples

```
>>> G=nx.Graph()
>>> G.add_nodes_from([0,1],color='red')
>>> G.add_nodes_from([2,3],color='blue')
>>> G.add_edges_from([(0,1),(2,3)])
>>> print nx.attribute_assortativity(G,'color')
1.0
```

networkx.numeric_assortativity**numeric_assortativity**(*G*, *attribute*)

Compute assortativity for numerical node attributes.

Assortativity measures the similarity of connections in the graph with respect to the given numeric attribute.

Parameters **G** : NetworkX graph**attribute** : string

Node attribute key

Returns **a**: float :

Assortativity of given attribute

Notes

This computes Eq. (21) in Ref. [R88], where e is the joint probability distribution (mixing matrix) of the specified attribute.

References

[R88]

Examples

```
>>> G=nx.Graph()
>>> G.add_nodes_from([0,1],size=2)
>>> G.add_nodes_from([2,3],size=3)
>>> G.add_edges_from([(0,1),(2,3)])
>>> print nx.numeric_assortativity(G,'size')
1.0
```

networkx.neighbor_connectivity

neighbor_connectivity(*G*)

Compute neighbor connectivity of graph.

The neighbor connectivity is the average nearest neighbor degree of a node of degree k .

Parameters *G* : NetworkX graph

Returns *d*: dictionary :

A dictionary keyed by degree k with the value of average neighbor degree.

Examples

```
>>> G=nx.cycle_graph(4)
>>> nx.neighbor_connectivity(G)
{2: 2.0}
```

```
>>> G=nx.complete_graph(4)
>>> nx.neighbor_connectivity(G)
{3: 3.0}
```

networkx.degree_pearsonr

degree_pearsonr(*G*)

Compute degree assortativity of graph.

Assortativity measures the similarity of connections in the graph with respect to the node degree.

Parameters *G* : NetworkX graph

Returns *r* : float

Assortativity of graph by degree.

Notes

This calls `scipy.stats.pearsonr()`.

References

[R50]

Examples

```
>>> G=nx.path_graph(4)
>>> r=nx.degree_pearsonr(G) # r=-0.5
```

4.18.2 Mixing

<code>attribute_mixing_matrix(G, attribute[, ...])</code>	Return mixing matrix for attribute.
<code>degree_mixing_matrix(G[, normalized])</code>	Return mixing matrix for attribute.
<code>degree_mixing_dict(G[, normalized])</code>	Return dictionary representation of mixing matrix for degree.
<code>attribute_mixing_dict(G, attribute[, normalized])</code>	Return dictionary representation of mixing matrix for attribute.

`networkx.attribute_mixing_matrix`

`attribute_mixing_matrix` (*G*, *attribute*, *mapping=None*, *normalized=True*)

Return mixing matrix for attribute.

Parameters **G** : graph

NetworkX graph object.

attribute : string

Node attribute key.

mapping : dictionary, optional

Mapping from node attribute to integer index in matrix. If not specified, an arbitrary ordering will be used.

normalized : bool (default=False)

Return counts if False or probabilities if True.

Returns **m**: numpy array :

Counts or joint probability of occurrence of attribute pairs.

`networkx.degree_mixing_matrix`

degree_mixing_matrix (*G*, *normalized=True*)

Return mixing matrix for attribute.

Parameters **G** : graph

NetworkX graph object.

normalized : bool (default=False)

Return counts if False or probabilities if True.

Returns **m**: numpy array :

Counts, or joint probability, of occurrence of node degree.

`networkx.degree_mixing_dict`

degree_mixing_dict (*G*, *normalized=False*)

Return dictionary representation of mixing matrix for degree.

Parameters **G** : graph

NetworkX graph object.

normalized : bool (default=False)

Return counts if False or probabilities if True.

Returns **d**: dictionary :

Counts or joint probability of occurrence of degree pairs.

`networkx.attribute_mixing_dict`

attribute_mixing_dict (*G*, *attribute*, *normalized=False*)

Return dictionary representation of mixing matrix for attribute.

Parameters **G** : graph

NetworkX graph object.

attribute : string

Node attribute key.

normalized : bool (default=False)

Return counts if False or probabilities if True.

Returns **d** : dictionary

Counts or joint probability of occurrence of attribute pairs.

Examples

```
>>> G=nx.Graph()
>>> G.add_nodes_from([0,1],color='red')
>>> G.add_nodes_from([2,3],color='blue')
>>> G.add_edge(1,3)
```

```

>>> d=nx.attribute_mixing_dict(G,'color')
>>> print d['red']['blue']
1
>>> print d['blue']['red'] # d symmetric for undirected graphs
1

```

4.19 Minimum Spanning Tree

Computes minimum spanning tree of a weighted graph.

`minimum_spanning_tree(G)` Generate a minimum spanning tree of an undirected weighted graph.

4.19.1 networkx.minimum_spanning_tree

`minimum_spanning_tree(G)`

Generate a minimum spanning tree of an undirected weighted graph.

A minimum spanning tree is a subgraph of the graph (a tree) with the minimum sum of edge weights.

Parameters **G** : NetworkX Graph

Returns **edges** : iterator

A generator that produces edges in the minimum spanning tree. The edges are three-tuples (u,v,w) where w is the weight.

Notes

Uses Kruskal's algorithm.

If the graph edges do not have a weight attribute a default weight of 1 will be assigned.

Modified code from David Eppstein, April 2006 <http://www.ics.uci.edu/~eppstein/PADS/>

Examples

```

>>> G=nx.cycle_graph(4)
>>> G.add_edge(0,3,weight=2) # assign weight 2 to edge 0-3
>>> mst=nx.minimum_spanning_tree(G) # a generator of MST edges
>>> edgelist=list(mst) # make a list of the edges
>>> print sorted(edgelist)
[(0, 1, {'weight': 1}), (1, 2, {'weight': 1}), (2, 3, {'weight': 1})]
>>> T=nx.Graph(edgelist) # build a graph of the MST.
>>> print sorted(T.edges(data=True))
[(0, 1, {'weight': 1}), (1, 2, {'weight': 1}), (2, 3, {'weight': 1})]

```

4.20 Operators

Operations on graphs including union, intersection, difference, complement, subgraph.

<code>cartesian_product(G, H[, create_using])</code>	Return the Cartesian product of G and H.
<code>compose(G, H[, create_using, name])</code>	Return a new graph of G composed with H.
<code>complement(G[, create_using, name])</code>	Return graph complement of G.
<code>union(G, H[, create_using, rename, name])</code>	Return the union of graphs G and H.
<code>disjoint_union(G, H)</code>	Return the disjoint union of graphs G and H, forcing distinct integer
<code>intersection(G, H[, create_using])</code>	Return a new graph that contains only the edges that exist in
<code>difference(G, H[, create_using])</code>	Return a new graph that contains the edges that exist in
<code>symmetric_difference(G, H[, create_using])</code>	Return new graph with edges that exist in in either G or H but

4.20.1 networkx.cartesian_product

cartesian_product (*G, H, create_using=None*)

Return the Cartesian product of G and H.

Parameters **G,H** : graph

A NetworkX graph

create_using : NetworkX graph

Use specified graph for result. Otherwise a new graph is created with the same type as G.

Notes

Only tested with Graph class. Graph, node, and edge attributes are not copied to the new graph.

4.20.2 networkx.compose

compose (*G, H, create_using=None, name=None*)

Return a new graph of G composed with H.

Composition is the simple union of the node sets and edge sets. The node sets of G and H need not be disjoint.

Parameters **G,H** : graph

A NetworkX graph

create_using : NetworkX graph

Use specified graph for result. Otherwise a new graph is created with the same type as G

name : string

Specify name for new graph

Notes

A new graph is returned, of the same class as G. It is recommended that G and H be either both directed or both undirected. Attributes from G take precedent over attributes from H.

4.20.3 networkx.complement

complement (*G*, *create_using=None*, *name=None*)
Return graph complement of *G*.

Parameters **G** : graph

A NetworkX graph

create_using : NetworkX graph

Use specified graph for result. Otherwise a new graph is created.

name : string

Specify name for new graph

Notes

Note that complement() does not create self-loops and also does not produce parallel edges for MultiGraphs. Graph, node, and edge data are not propagated to the new graph.

4.20.4 networkx.union

union (*G*, *H*, *create_using=None*, *rename=False*, *name=None*)
Return the union of graphs *G* and *H*.

Graphs *G* and *H* must be disjoint, otherwise an exception is raised.

Parameters **G,H** : graph

A NetworkX graph

create_using : NetworkX graph

Use specified graph for result. Otherwise a new graph is created with the same type as *G*.

rename : bool (default=False)

Node names of *G* and *H* can be changed by specifying the tuple *rename*=(*'G-'*,*'H-'*) (for example). Node *u* in *G* is then renamed "*G-u*" and *v* in *H* is renamed "*H-v*".

name : string

Specify the name for the union graph

See Also:

`disjoint_union`

Notes

To force a disjoint union with node relabeling, use `disjoint_union(G,H)` or `convert_node_labels_to_integers()`.

Graph, edge, and node attributes are propagated from *G* and *H* to the union graph. If a graph attribute is present in both *G* and *H* the value from *G* is used.

4.20.5 networkx.disjoint_union

disjoint_union (*G, H*)

Return the disjoint union of graphs G and H, forcing distinct integer node labels.

Parameters **G,H** : graph

A NetworkX graph

Notes

A new graph is created, of the same class as G. It is recommended that G and H be either both directed or both undirected.

4.20.6 networkx.intersection

intersection (*G, H, create_using=None*)

Return a new graph that contains only the edges that exist in both G and H.

The node sets of H and G must be the same.

Parameters **G,H** : graph

A NetworkX graph. G and H must have the same node sets.

create_using : NetworkX graph

Use specified graph for result. Otherwise a new graph is created with the same type as G.

Notes

Attributes from the graph, nodes, and edges are not copied to the new graph. If you want a new graph of the intersection of G and H with the attributes (including edge data) from G use `remove_nodes_from()` as follows

```
>>> G=nx.path_graph(3)
>>> H=nx.path_graph(5)
>>> R=G.copy()
>>> R.remove_nodes_from(n for n in G if n not in H)
```

4.20.7 networkx.difference

difference (*G, H, create_using=None*)

Return a new graph that contains the edges that exist in in G but not in H.

The node sets of H and G must be the same.

Parameters **G,H** : graph

A NetworkX graph. G and H must have the same node sets.

create_using : NetworkX graph

Use specified graph for result. Otherwise a new graph is created with the same type as G.

Notes

Attributes from the graph, nodes, and edges are not copied to the new graph. If you want a new graph of the difference of G and H with with the attributes (including edge data) from G use `remove_nodes_from()` as follows

```
>>> G=nx.path_graph(3)
>>> H=nx.path_graph(5)
>>> R=G.copy()
>>> R.remove_nodes_from(n for n in G if n in H)
```

4.20.8 networkx.symmetric_difference

symmetric_difference (*G, H, create_using=None*)

Return new graph with edges that exist in either G or H but not both.

The node sets of H and G must be the same.

Parameters **G,H** : graph

A NetworkX graph. G and H must have the same node sets.

create_using : NetworkX graph

Use specified graph for result. Otherwise a new graph is created with the same type as G.

Notes

Attributes from the graph, nodes, and edges are not copied to the new graph.

4.21 Shortest Paths

Compute the shortest paths and path lengths between nodes in the graph.

These algorithms work with undirected and directed graphs.

For directed graphs the paths can be computed in the reverse order by first flipping the edge orientation using `R=G.reverse(copy=False)`.

<code>shortest_path(G[, source, target, weighted])</code>	Compute shortest paths in the graph.
<code>shortest_path_length(G[, source, target, ...])</code>	Compute shortest path lengths in the graph.
<code>average_shortest_path_length(G[, weighted])</code>	Return the average shortest path length.

4.21.1 networkx.shortest_path

shortest_path (*G, source=None, target=None, weighted=False*)

Compute shortest paths in the graph.

Parameters **G** : NetworkX graph

source : node, optional

Starting node for path. If not specified compute shortest paths for all connected node pairs.

target : node, optional

Ending node for path. If not specified compute shortest paths for every node reachable from the source.

weighted : bool, optional

If True consider weighted edges when finding shortest path.

Returns path: list or dictionary :

If the source and target are both specified return a single list of nodes in a shortest path. If only the source is specified return a dictionary keyed by targets with a list of nodes in a shortest path. If neither the source or target is specified return a dictionary of dictionaries with path[source][target]=[list of nodes in path].

Notes

There may be more than one shortest path between a source and target. This returns only one of them.

If weighted=True and the graph has no 'weight' edge attribute the value 1 will be used.

For digraphs this returns a shortest directed path. To find paths in the reverse direction use `G.reverse(copy=False)` first to flip the edge orientation.

Examples

```
>>> G=nx.path_graph(5)
>>> print nx.shortest_path(G,source=0,target=4)
[0, 1, 2, 3, 4]
>>> p=nx.shortest_path(G,source=0) # target not specified
>>> p[4]
[0, 1, 2, 3, 4]
>>> p=nx.shortest_path(G) # source,target not specified
>>> p[0][4]
[0, 1, 2, 3, 4]
```

4.21.2 networkx.shortest_path_length

shortest_path_length (*G*, *source=None*, *target=None*, *weighted=False*)

Compute shortest path lengths in the graph.

This function can compute the single source shortest path lengths by specifying only the source or all pairs shortest path lengths by specifying neither the source or target.

Parameters G : NetworkX graph

source : node, optional

Starting node for path. If not specified compute shortest path lengths for all connected node pairs.

target : node, optional

Ending node for path. If not specified compute shortest path lengths for every node reachable from the source.

weighted : bool, optional

If True consider weighted edges when finding shortest path length.

Returns `length` : number, or container of numbers

If the source and target are both specified return a single number for the shortest path. If only the source is specified return a dictionary keyed by targets with a the shortest path as keys. If neither the source or target is specified return a dictionary of dictionaries with `length[source][target]=value`.

Raises `NetworkXError` :

If no path exists between source and target.

Notes

If `weighted=True` and the graph has no 'weight' edge attribute the value 1 will be used.

For digraphs this returns the shortest directed path. To find path lengths in the reverse direction use `G.reverse(copy=False)` first to flip the edge orientation.

Examples

```
>>> G=nx.path_graph(5)
>>> print nx.shortest_path_length(G, source=0, target=4)
4
>>> p=nx.shortest_path_length(G, source=0) # target not specified
>>> p[4]
4
>>> p=nx.shortest_path_length(G) # source, target not specified
>>> p[0][4]
4
```

4.21.3 networkx.average_shortest_path_length

average_shortest_path_length (*G*, *weighted=False*)

Return the average shortest path length.

The average shortest path length is the sum of path lengths $d(u,v)$ between all pairs of nodes (assuming the length is zero if v is not reachable from v) normalized by $n*(n-1)$ where n is the number of nodes in G .

Parameters **G** : NetworkX graph

weighted : bool, optional, default=False

If True use edge weights on path.

Notes

If `weighted=True` and the graph has no 'weight' edge attribute the value 1 will be used.

Examples

```
>>> G=nx.path_graph(5)
>>> print nx.average_shortest_path_length(G)
2.0
```

4.21.4 Advanced Interface

Shortest path algorithms for unweighted graphs.

<code>single_source_shortest_path(G, source[, cutoff])</code>	Compute shortest path between source and all other nodes reachable from source.
<code>single_source_shortest_path_length(G, source)</code>	Compute the shortest path lengths from source to all reachable nodes.
<code>all_pairs_shortest_path(G[, cutoff])</code>	Compute shortest paths between all nodes.
<code>all_pairs_shortest_path_length(G[, cutoff])</code>	Compute the shortest path lengths between all nodes in G.
<code>predecessor(G, source[, target, cutoff, ...])</code>	Returns dictionary of predecessors for the path from source to all
<code>floyd_warshall(G)</code>	The Floyd-Warshall algorithm for all pairs shortest paths.

networkx.single_source_shortest_path

single_source_shortest_path (*G*, *source*, *cutoff=None*)

Compute shortest path between source and all other nodes reachable from source.

Parameters **G** : NetworkX graph

source : node label

Starting node for path

cutoff : integer, optional

Depth to stop the search. Only paths of length \leq cutoff are returned.

Returns **lengths** : dictionary

Dictionary, keyed by target, of shortest paths.

See Also:

`shortest_path`

Notes

There may be more than one shortest path between the source and target nodes. This function returns only one of them.

Examples

```
>>> G=nx.path_graph(5)
>>> path=nx.single_source_shortest_path(G,0)
>>> path[4]
[0, 1, 2, 3, 4]
```

networkx.single_source_shortest_path_length

single_source_shortest_path_length(*G*, *source*, *cutoff=None*)

Compute the shortest path lengths from source to all reachable nodes.

Parameters *G* : NetworkX graph

source : node

Starting node for path

cutoff : integer, optional

Depth to stop the search. Only paths of length \leq cutoff are returned.

Returns **lengths** : dictionary

Dictionary of shortest path lengths keyed by target.

See Also:

[shortest_path_length](#)

Examples

```
>>> G=nx.path_graph(5)
>>> length=nx.single_source_shortest_path_length(G,0)
>>> length[4]
4
>>> print length
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4}
```

networkx.all_pairs_shortest_path

all_pairs_shortest_path(*G*, *cutoff=None*)

Compute shortest paths between all nodes.

Parameters *G* : NetworkX graph

cutoff : integer, optional

Depth to stop the search. Only paths of length \leq cutoff are returned.

Returns **lengths** : dictionary

Dictionary, keyed by source and target, of shortest paths.

See Also:

[floyd_warshall](#)

Examples

```
>>> G=nx.path_graph(5)
>>> path=nx.all_pairs_shortest_path(G)
>>> print path[0][4]
[0, 1, 2, 3, 4]
```

networkx.all_pairs_shortest_path_length

all_pairs_shortest_path_length (*G*, *cutoff=None*)

Compute the shortest path lengths between all nodes in *G*.

Parameters *G* : NetworkX graph

cutoff : integer, optional

depth to stop the search. Only paths of length \leq cutoff are returned.

Returns **lengths** : dictionary

Dictionary of shortest path lengths keyed by source and target.

Notes

The dictionary returned only has keys for reachable node pairs.

Examples

```
>>> G=nx.path_graph(5)
>>> length=nx.all_pairs_shortest_path_length(G)
>>> print length[1][4]
3
>>> length[1]
{0: 1, 1: 0, 2: 1, 3: 2, 4: 3}
```

networkx.predecessor

predecessor (*G*, *source*, *target=None*, *cutoff=None*, *return_seen=None*)

Returns dictionary of predecessors for the path from source to all nodes in *G*.

Parameters *G* : NetworkX graph

source : node label

Starting node for path

target : node label, optional

Ending node for path. If provided only predecessors between source and target are returned

cutoff : integer, optional

Depth to stop the search. Only paths of length \leq cutoff are returned.

Returns pred : dictionary

Dictionary, keyed by node, of predecessors in the shortest path.

Examples

```
>>> G=nx.path_graph(4)
>>> print G.nodes()
[0, 1, 2, 3]
>>> nx.predecessor(G,0)
{0: [], 1: [0], 2: [1], 3: [2]}
```

networkx.floyd_warshall

floyd_warshall (*G*)

The Floyd-Warshall algorithm for all pairs shortest paths.

Parameters G : NetworkX graph

Returns distance,pred : dictionaries

A dictionary, keyed by source and target, of shortest path distance and predecessors in the shortest path.

See Also:

`all_pairs_shortest_path`, `all_pairs_shortest_path_length`

Notes

This algorithm is most appropriate for dense graphs. The running time is $O(n^3)$, and running space is $O(n^2)$ where n is the number of nodes in G .

Shortest path algorithms for weighed graphs.

<code>dijkstra_path(G, source, target[, weight])</code>	Returns the shortest path from source to target in a weighted graph.
<code>dijkstra_path_length(G, source, target[, weight])</code>	Returns the shortest path length from source to target in a weighted graph.
<code>single_source_dijkstra_path(G, source[, weight])</code>	Compute shortest path between source and all other reachable nodes for a weighted graph.
<code>single_source_dijkstra_path_length(G, source)</code>	Compute shortest path length between source and all other reachable nodes for a weighted graph.
<code>all_pairs_dijkstra_path(G[, weight])</code>	Compute shortest paths between all nodes in a weighted graph.
<code>all_pairs_dijkstra_path_length(G[, weight])</code>	Compute shortest path lengths between all nodes in a weighted graph.
<code>single_source_dijkstra(G, source[, target, ...])</code>	Compute shortest paths and lengths in a weighted graph G .
<code>bidirectional_dijkstra(G, source, target[, ...])</code>	Dijkstra's algorithm for shortest paths using bidirectional search.
<code>bidirectional_shortest_path(G, source, target)</code>	Return a list of nodes in a shortest path between source and target.
<code>dijkstra_predecessor_and_distance(G, source)</code>	Compute shortest path length and predecessors on shortest paths in weighted graphs.

networkx.dijkstra_path

`dijkstra_path` (*G*, *source*, *target*, *weight='weight'*)

Returns the shortest path from source to target in a weighted graph G.

Parameters **G** : NetworkX graph

source : node

Starting node

target : node

Ending node

weight: string, optional :

Edge data key corresponding to the edge weight

Returns **path** : list

List of nodes in a shortest path.

See Also:

`bidirectional_dijkstra`

Notes

Uses a bidirectional version of Dijkstra's algorithm. Edge weight attributes must be numerical.

Examples

```
>>> G=nx.path_graph(5)
>>> print nx.dijkstra_path(G,0,4)
[0, 1, 2, 3, 4]
```

networkx.dijkstra_path_length

`dijkstra_path_length` (*G*, *source*, *target*, *weight='weight'*)

Returns the shortest path length from source to target in a weighted graph G.

Parameters **G** : NetworkX graph, weighted

source : node label

starting node for path

target : node label

ending node for path

weight: string, optional :

Edge data key corresponding to the edge weight

Returns **length** : number

Shortest path length.

Raises **NetworkXError** :

If no path exists between source and target.

See Also:

`bidirectional_dijkstra`

Notes

Edge weight attributes must be numerical.

Examples

```
>>> G=nx.path_graph(5) # a weighted graph by default
>>> print nx.dijkstra_path_length(G,0,4)
4
```

`networkx.single_source_dijkstra_path`

`single_source_dijkstra_path` (*G*, *source*, *weight='weight'*)

Compute shortest path between source and all other reachable nodes for a weighted graph.

Parameters *G* : NetworkX graph

source : node

Starting node for path.

weight: string, optional :

Edge data key corresponding to the edge weight

Returns *paths* : dictionary

Dictionary of shortest path lengths keyed by target.

See Also:

`single_source_dijkstra`

Notes

Edge weight attributes must be numerical.

Examples

```
>>> G=nx.path_graph(5)
>>> path=nx.single_source_dijkstra_path(G,0)
>>> path[4]
[0, 1, 2, 3, 4]
```

`networkx.single_source_dijkstra_path_length`

`single_source_dijkstra_path_length` (*G*, *source*, *weight='weight'*)

Compute shortest path length between source and all other reachable nodes for a weighted graph.

Parameters *G* : NetworkX graph

source : node label

Starting node for path

weight: string, optional :

Edge data key corresponding to the edge weight

Returns *paths* : dictionary

Dictionary of shortest paths keyed by target.

See Also:

`single_source_dijkstra`

Notes

Edge data must be numerical values for XGraph and XDiGraphs.

Examples

```
>>> G=nx.path_graph(5)
>>> length=nx.single_source_dijkstra_path_length(G,0)
>>> length[4]
4
>>> print length
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4}
```

`networkx.all_pairs_dijkstra_path`

`all_pairs_dijkstra_path` (*G*, *weight='weight'*)

Compute shortest paths between all nodes in a weighted graph.

Parameters *G* : NetworkX graph

weight: string, optional :

Edge data key corresponding to the edge weight

Returns *distance* : dictionary

Dictionary, keyed by source and target, of shortest paths.

See Also:

`floyd_warshall`

Examples

```
>>> G=nx.path_graph(5)
>>> path=nx.all_pairs_dijkstra_path(G)
>>> print path[0][4]
[0, 1, 2, 3, 4]
```

networkx.all_pairs_dijkstra_path_length

all_pairs_dijkstra_path_length (*G*, *weight='weight'*)

Compute shortest path lengths between all nodes in a weighted graph.

Parameters *G* : NetworkX graph

weight: string, optional :

Edge data key corresponding to the edge weight

Returns *distance* : dictionary

Dictionary, keyed by source and target, of shortest path lengths.

Notes

The dictionary returned only has keys for reachable node pairs.

Examples

```
>>> G=nx.path_graph(5)
>>> length=nx.all_pairs_dijkstra_path_length(G)
>>> print length[1][4]
3
>>> length[1]
{0: 1, 1: 0, 2: 1, 3: 2, 4: 3}
```

networkx.single_source_dijkstra

single_source_dijkstra (*G*, *source*, *target=None*, *cutoff=None*, *weight='weight'*)

Compute shortest paths and lengths in a weighted graph *G*.

Uses Dijkstra's algorithm for shortest paths.

Parameters *G* : NetworkX graph

source : node label

Starting node for path

target : node label, optional

Ending node for path

cutoff : integer or float, optional

Depth to stop the search. Only paths of length \leq cutoff are returned.

Returns `distance,path` : dictionaries

Returns a tuple of two dictionaries keyed by node. The first dictionary stores distance from the source. The second stores the path from the source to that node.

See Also:

`single_source_dijkstra_path`, `single_source_dijkstra_path_length`

Notes

Distances are calculated as sums of weighted edges traversed. Edges must hold numerical values for Graph and DiGraphs.

Based on the Python cookbook recipe (119466) at <http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/119466>

This algorithm is not guaranteed to work if edge weights are negative or are floating point numbers (overflows and roundoff errors can cause problems).

Examples

```
>>> G=nx.path_graph(5)
>>> length,path=nx.single_source_dijkstra(G,0)
>>> print length[4]
4
>>> print length
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4}
>>> path[4]
[0, 1, 2, 3, 4]
```

`networkx.bidirectional_dijkstra`

`bidirectional_dijkstra` (*G*, *source*, *target*, *weight='weight'*)

Dijkstra's algorithm for shortest paths using bidirectional search.

Parameters **G** : NetworkX graph

source : node

Starting node.

target : node

Ending node.

weight: string, optional :

Edge data key corresponding to the edge weight

Returns **length** : number

Shortest path length.

Returns a tuple of two dictionaries keyed by node. :

The first dictionary stores distance from the source. :

The second stores the path from the source to that node. :

Raise an exception if no path exists. :

Raises NetworkXError :

If no path exists between source and target.

See Also:

`shortest_path`, `shortest_path_length`

Notes

Edge weight attributes must be numerical. Distances are calculated as sums of weighted edges traversed.

In practice bidirectional Dijkstra is much more than twice as fast as ordinary Dijkstra.

Ordinary Dijkstra expands nodes in a sphere-like manner from the source. The radius of this sphere will eventually be the length of the shortest path. Bidirectional Dijkstra will expand nodes from both the source and the target, making two spheres of half this radius. Volume of the first sphere is $\pi*r*r$ while the others are $2*\pi*r/2*r/2$, making up half the volume.

This algorithm is not guaranteed to work if edge weights are negative or are floating point numbers (overflows and roundoff errors can cause problems).

Examples

```
>>> G=nx.path_graph(5)
>>> length,path=nx.bidirectional_dijkstra(G,0,4)
>>> print length
4
>>> print path
[0, 1, 2, 3, 4]
```

networkx.bidirectional_shortest_path

bidirectional_shortest_path(*G*, *source*, *target*)

Return a list of nodes in a shortest path between source and target.

Parameters **G** : NetworkX graph

source : node label

starting node for path

target : node label

ending node for path

Returns **path: list** :

List of nodes in a path from source to target.

See Also:

`shortest_path`

Notes

This algorithm is used by `shortest_path(G,source,target)`.

networkx.dijkstra_predecessor_and_distance

dijkstra_predecessor_and_distance (*G*, *source*, *weight='weight'*)

Compute shorest path length and predecessors on shortest paths in weighted graphs.

Parameters **G** : NetworkX graph

source : node label

Starting node for path

weight: string, optional :

Edge data key corresponding to the edge weight

Returns **pred,distance** : dictionaries

Returns two dictionaries representing a list of predecessors of a node and the distance to each node.

Notes

The list of predecessors contains more than one element only when there are more than one shortest paths to the key node.

4.21.5 A* Algorithm

Shortest paths and path lengths using A* (“A star”) algorithm.

<code>astar_path(G, source, target[, heuristic])</code>	Return a list of nodes in a shortest path between source and target
<code>astar_path_length(G, source, target[, heuristic])</code>	Return a list of nodes in a shortest path between source and target

networkx.astar_path

astar_path (*G*, *source*, *target*, *heuristic=None*)

Return a list of nodes in a shortest path between source and target using the A* (“A-star”) algorithm.

There may be more than one shortest path. This returns only one.

Parameters **G** : NetworkX graph

source : node

Starting node for path

target : node

Ending node for path

heuristic : function

A function to evaluate the estimate of the distance from the a node to the target. The function takes two nodes arguments and must return a number.

See Also:

`shortest_path`, `dijkstra_path`

Examples

```
>>> G=nx.path_graph(5)
>>> print nx.astar_path(G,0,4)
[0, 1, 2, 3, 4]
>>> G=nx.grid_graph(dim=[3,3]) # nodes are two-tuples (x,y)
>>> def dist((x1, y1), (x2, y2)):
...     return ((x1 - x2) ** 2 + (y1 - y2) ** 2) ** 0.5
>>> print nx.astar_path(G, (0,0), (2,2), dist)
[(0, 0), (0, 1), (1, 1), (1, 2), (2, 2)]
```

networkx.astar_path_length

astar_path_length(*G*, *source*, *target*, *heuristic=None*)

Return a list of nodes in a shortest path between source and target using the A* (“A-star”) algorithm.

Parameters **G** : NetworkX graph

source : node

Starting node for path

target : node

Ending node for path

heuristic : function

A function to evaluate the estimate of the distance from the a node to the target. The function takes two nodes arguments and must return a number.

See Also:

`astar_path`

4.22 Traversal

4.22.1 Depth First Search

Search algorithms.

<code>dfs_preorder(G[, source, reverse_graph])</code>	Return list of nodes connected to source in depth-first-search preorder.
<code>dfs_postorder(G[, source, reverse_graph])</code>	Return list of nodes connected to source in depth-first-search postorder.
<code>dfs_predecessor(G[, source, reverse_graph])</code>	Return predecessors of depth-first-search with root at source.
<code>dfs_successor(G[, source, reverse_graph])</code>	Return successors of depth-first-search with root at source.
<code>dfs_tree(G[, source, reverse_graph])</code>	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 successors 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).

4.23 Vitality

Vitality measures.

`closeness_vitality(G[, v, weighted_edges])` Compute closeness vitality for nodes.

4.23.1 networkx.closeness_vitality

closeness_vitality (*G*, *v=None*, *weighted_edges=False*)

Compute closeness vitality for nodes.

Closeness vitality at a node is the change in the sum of distances between all node pairs when excluding a that node.

Parameters **G** : graph

A networkx graph

v : node, optional

Return only the value for node *v*.

weighted_edges : bool, optional

Consider the edge weights in determining the shortest paths. If False, all edge weights are considered equal.

Returns `nodes` : dictionary

Dictionary with nodes as keys and closeness vitality as the value.

See Also:

`closeness_centrality`

Examples

```
>>> G=nx.cycle_graph(3)
>>> nx.closeness_vitality(G)
{0: 4.0, 1: 4.0, 2: 4.0}
```


FUNCTIONS

Functional interface to graph methods and assorted utilities.

5.1 Graph functions

<code>density(G)</code>	Return the density of a graph.
<code>info(G[, n, output_to_file])</code>	Print short summary of information for graph G or node n.
<code>degree_histogram(G)</code>	Return a list of the frequency of each degree value.
<code>freeze(G)</code>	Modify graph to prevent addition of nodes or edges.
<code>is_frozen(G)</code>	Return True if graph is frozen.
<code>create_empty_copy(G[, with_nodes])</code>	Return a copy of the graph G with all of the edges removed.

5.1.1 networkx.density

density(G)

Return the density of a graph.

The density for undirected graphs is

$$d = \frac{2m}{n(n-1)},$$

and for directed graphs is

$$d = \frac{m}{n(n-1)},$$

where n is the number of nodes and m is the number of edges in G .

Notes

The density is 0 for an graph without edges and 1.0 for a complete graph.

The density of multigraphs can be higher than 1.

5.1.2 networkx.info

info(G, n=None, output_to_file=None)

Print short summary of information for graph G or node n.

Parameters **G** : Networkx graph

A graph

n : node (any hashable)

A node from the graph G

output_to_file: filehandle, optional (default= standard output) :

5.1.3 networkx.degree_histogram

degree_histogram(G)

Return a list of the frequency of each degree value.

Parameters **G** : Networkx graph

A graph

Returns **hist** : list

A list of frequencies of degrees. The degree values are the index in the list.

Notes

Note: the bins are width one, hence len(list) can be large (Order(number_of_edges))

5.1.4 networkx.freeze

freeze(G)

Modify graph to prevent addition of nodes or edges.

Parameters **G** : graph

A NetworkX graph

See Also:

`is_frozen`

Notes

This does not prevent modification of edge data.

To “unfreeze” a graph you must make a copy.

Examples

```
>>> G=nx.path_graph(4)
>>> G=nx.freeze(G)
>>> G.add_edge(4,5)
...
NetworkXError: Frozen graph can't be modified
```

5.1.5 networkx.is_frozen

is_frozen (*G*)

Return True if graph is frozen.

Parameters **G** : graph

A NetworkX graph

See Also:

`freeze`

5.1.6 networkx.create_empty_copy

create_empty_copy (*G*, *with_nodes=True*)

Return a copy of the graph *G* with all of the edges removed.

Parameters **G** : graph

A NetworkX graph

with_nodes : bool (default=True)

Include nodes.

Notes

Graph, node, and edge data is not propagated to the new graph.

GRAPH GENERATORS

6.1 Atlas

Generators for the small graph atlas.

See “An Atlas of Graphs” by Ronald C. Read and Robin J. Wilson, Oxford University Press, 1998.

Because of its size, this module is not imported by default.

`graph_atlas_g()` Return the list `[G0,G1,...,G1252]` of graphs as named in the Graph Atlas.

6.1.1 `networkx.generators.atlas.graph_atlas_g`

`graph_atlas_g()`

Return the list `[G0,G1,...,G1252]` of graphs as named in the Graph Atlas. `G0,G1,...,G1252` are all graphs with up to 7 nodes.

The graphs are listed:

1. in increasing order of number of nodes;
2. for a fixed number of nodes, in increasing order of the number of edges;
3. for fixed numbers of nodes and edges, in increasing order of the degree sequence, for example `111223 < 112222`;
4. for fixed degree sequence, in increasing number of automorphisms.

Note that indexing is set up so that for `GAG=graph_atlas_g()`, then `G123=GAG[123]` and `G[0]=empty_graph(0)`

6.2 Classic

Generators for some classic graphs.

The typical graph generator is called as follows:

```
>>> G=nx.complete_graph(100)
```

returning the complete graph on `n` nodes labeled `0,...,99` as a simple graph. Except for `empty_graph`, all the generators in this module return a Graph class (i.e. a simple, undirected graph).

<code>balanced_tree(r, h[, create_using])</code>	Return the perfectly balanced r-tree of height h.
<code>barbell_graph(m1, m2[, create_using])</code>	Return the Barbell Graph: two complete graphs connected by a path.
<code>complete_graph(n[, create_using])</code>	Return the Complete graph K_n with n nodes.
<code>complete_bipartite_graph(n1, n2[, create_using])</code>	Return the complete bipartite graph $K_{\{n1_n2\}}$.
<code>circular_ladder_graph(n[, create_using])</code>	Return the circular ladder graph CL_n of length n.
<code>cycle_graph(n[, create_using])</code>	Return the cycle graph C_n over n nodes.
<code>dorogovtsev_goltsev_mendes_graph(n[, create_using])</code>	Return the hierarchically constructed Dorogovtsev-Goltsev-Mendes graph.
<code>empty_graph(n[, create_using])</code>	Return the empty graph with n nodes and zero edges.
<code>grid_2d_graph(m, n[, periodic, create_using])</code>	Return the 2d grid graph of $m \times n$ nodes, each connected to its nearest neighbors.
<code>grid_graph(dim[, periodic, create_using])</code>	Return the n-dimensional grid graph.
<code>hypercube_graph(n[, create_using])</code>	Return the n-dimensional hypercube.
<code>ladder_graph(n[, create_using])</code>	Return the Ladder graph of length n.
<code>lollipop_graph(m, n[, create_using])</code>	Return the Lollipop Graph; K_m connected to P_n .
<code>null_graph([create_using])</code>	Return the Null graph with no nodes or edges.
<code>path_graph(n[, create_using])</code>	Return the Path graph P_n of n nodes linearly connected
<code>star_graph(n[, create_using])</code>	Return the Star graph with $n+1$ nodes:
<code>trivial_graph([create_using])</code>	Return the Trivial graph with one node (with integer label 0)
<code>wheel_graph(n[, create_using])</code>	Return the wheel graph: a single hub node connected to each node of the $(n-1)$ -node cycle graph.

6.2.1 networkx.generators.classic.balanced_tree

balanced_tree (*r, h, create_using=None*)

Return the perfectly balanced r-tree of height h.

For $r \geq 2$, $h \geq 1$, 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 $r+1$.

$\text{number_of_nodes} = 1 + r + r^2 + \dots + r^h = (r^{h+1} - 1) / (r - 1)$, $\text{number_of_edges} = \text{number_of_nodes} - 1$.

Node labels are the integers 0 (the root) up to $\text{number_of_nodes} - 1$.

6.2.2 networkx.generators.classic.barbell_graph

barbell_graph (*m1, m2, create_using=None*)

Return the Barbell Graph: two complete graphs connected by a path.

For $m1 > 1$ and $m2 \geq 0$.

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, $m1, \dots, m1 + m2 - 1$ for the path, and $m1 + m2, \dots, 2 * m1 + m2 - 1$ 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 text on Random Walks on Graphs.

6.2.3 networkx.generators.classic.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.

6.2.4 networkx.generators.classic.complete_bipartite_graph

complete_bipartite_graph (*n1*, *n2*, *create_using=None*)

Return the complete bipartite graph $K_{\{n1, n2\}}$.

Composed of two partitions with *n1* nodes in the first and *n2* nodes in the second. Each node in the first is connected to each node in the second.

Node labels are the integers 0 to *n1*+*n2*-1

6.2.5 networkx.generators.classic.circular_ladder_graph

circular_ladder_graph (*n*, *create_using=None*)

Return the circular ladder graph CL_n of length *n*.

CL_n consists of two concentric *n*-cycles in which each of the *n* pairs of concentric nodes are joined by an edge.

Node labels are the integers 0 to *n*-1

6.2.6 networkx.generators.classic.cycle_graph

cycle_graph (*n*, *create_using=None*)

Return the cycle graph C_n over *n* nodes.

C_n is the *n*-path with two end-nodes connected.

Node labels are the integers 0 to *n*-1. If *create_using* is a DiGraph, the direction is in increasing order.

6.2.7 networkx.generators.classic.dorogovtsev_goltsev_mendes_graph

dorogovtsev_goltsev_mendes_graph (*n*, *create_using=None*)

Return the hierarchically constructed Dorogovtsev-Goltsev-Mendes graph.

n is the generation. See: [arXiv:/cond-mat/0112143](https://arxiv.org/abs/cond-mat/0112143) by Dorogovtsev, Goltsev and Mendes.

6.2.8 networkx.generators.classic.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)`.

6.2.9 `networkx.generators.classic.grid_2d_graph`

`grid_2d_graph` ($m, n, periodic=False, create_using=None$)

Return the 2d grid graph of $m \times n$ nodes, each connected to its nearest neighbors. Optional argument `periodic=True` will connect boundary nodes via periodic boundary conditions.

6.2.10 `networkx.generators.classic.grid_graph`

`grid_graph` ($dim, periodic=False, create_using=None$)

Return the n -dimensional grid graph.

The dimension is the length of the list 'dim' and the size in each dimension is the value of the list element.

E.g. `G=grid_graph(dim=[2,3])` produces a 2×3 grid graph.

If `periodic=True` then join grid edges with periodic boundary conditions.

6.2.11 `networkx.generators.classic.hypercube_graph`

`hypercube_graph` ($n, create_using=None$)

Return the n -dimensional hypercube.

Node labels are the integers 0 to $2^{*n} - 1$.

6.2.12 `networkx.generators.classic.ladder_graph`

`ladder_graph` ($n, create_using=None$)

Return the Ladder graph of length n .

This is two rows of n nodes, with each pair connected by a single edge.

Node labels are the integers 0 to $2 * n - 1$.

6.2.13 `networkx.generators.classic.lollipop_graph`

`lollipop_graph` ($m, n, create_using=None$)

Return the Lollipop Graph; K_m connected to P_n .

This is the Barbell Graph without the right barbell.

For $m > 1$ and $n \geq 0$, the complete graph K_m is connected to the path P_n . The resulting $m+n$ nodes are labelled $0, \dots, m-1$ for the complete graph and $m, \dots, m+n-1$ for the path. The 2 subgraphs are joined via the edge $(m-1, m)$. If $n=0$, this is merely a complete graph.

Node labels are the integers 0 to `number_of_nodes - 1`.

(This graph is an extremal example in David Aldous and Jim Fill's etext on Random Walks on Graphs.)

6.2.14 `networkx.generators.classic.null_graph`

`null_graph` (*create_using=None*)

Return the Null graph with no nodes or edges.

See `empty_graph` for the use of `create_using`.

6.2.15 `networkx.generators.classic.path_graph`

`path_graph` (*n, create_using=None*)

Return the Path graph P_n of n nodes linearly connected by $n-1$ edges.

Node labels are the integers 0 to $n - 1$. If `create_using` is a DiGraph then the edges are directed in increasing order.

6.2.16 `networkx.generators.classic.star_graph`

`star_graph` (*n, create_using=None*)

Return the Star graph with $n+1$ nodes: one center node, connected to n outer nodes.

Node labels are the integers 0 to n .

6.2.17 `networkx.generators.classic.trivial_graph`

`trivial_graph` (*create_using=None*)

Return the Trivial graph with one node (with integer label 0) and no edges.

6.2.18 `networkx.generators.classic.wheel_graph`

`wheel_graph` (*n, create_using=None*)

Return the wheel graph: a single hub node connected to each node of the $(n-1)$ -node cycle graph.

Node labels are the integers 0 to $n - 1$.

6.3 Small

Various small and named graphs, together with some compact generators.

<code>make_small_graph(graph_description[, ...])</code>	Return the small graph described by <code>graph_description</code> .
<code>LCF_graph(n, shift_list, repeats[, create_using])</code>	Return the cubic graph specified in LCF notation.
<code>bull_graph([create_using])</code>	Return the Bull graph.
<code>chvatal_graph([create_using])</code>	Return the Chvatal graph.
<code>cubical_graph([create_using])</code>	Return the 3-regular Platonic Cubical graph.
<code>desargues_graph([create_using])</code>	Return the Desargues graph.
<code>diamond_graph([create_using])</code>	Return the Diamond graph.
<code>dodecahedral_graph([create_using])</code>	Return the Platonic Dodecahedral graph.
<code>frucht_graph([create_using])</code>	Return the Frucht Graph.
<code>heawood_graph([create_using])</code>	Return the Heawood graph, a (3,6) cage.
<code>house_graph([create_using])</code>	Return the House graph (square with triangle on top).
<code>house_x_graph([create_using])</code>	Return the House graph with a cross inside the house square.
<code>icosahedral_graph([create_using])</code>	Return the Platonic Icosahedral graph.
<code>krackhardt_kite_graph([create_using])</code>	Return the Krackhardt Kite Social Network.
<code>moebius_kantor_graph([create_using])</code>	Return the Moebius-Kantor graph.
<code>octahedral_graph([create_using])</code>	Return the Platonic Octahedral graph.
<code>pappus_graph()</code>	Return the Pappus graph.
<code>petersen_graph([create_using])</code>	Return the Petersen graph.
<code>sedgewick_maze_graph([create_using])</code>	Return a small maze with a cycle.
<code>tetrahedral_graph([create_using])</code>	Return the 3-regular Platonic Tetrahedral graph.
<code>truncated_cube_graph([create_using])</code>	Return the skeleton of the truncated cube.
<code>truncated_tetrahedron_graph([create_using])</code>	Return the skeleton of the truncated Platonic tetrahedron.
<code>tutte_graph([create_using])</code>	Return the Tutte graph.

6.3.1 networkx.generators.small.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 `0,...,n-1`.

If `ltype=`“adjacencylist” then `xlist` is an adjacency list with exactly `n` entries, in with the `j`’th entry (which can be empty) specifies the nodes connected to vertex `j`. e.g. the “square” graph `C_4` can be obtained by

```
>>> G=nx.make_small_graph(["adjacencylist", "C_4", 4, [[2, 4], [1, 3], [2, 4], [1, 3]]])
```

or, since we do not need to add edges twice,

```
>>> G=nx.make_small_graph(["adjacencylist", "C_4", 4, [[2, 4], [3], [4], []]])
```

If `ltype=`“edgelist” then `xlist` is an edge list written as `[[v1,w2],[v2,w2],...,[vk,wk]]`, where `vj` and `wj` integers in the range `1,...,n` e.g. the “square” graph `C_4` can be obtained by

```
>>> G=nx.make_small_graph(["edgelist", "C_4", 4, [[1, 2], [3, 4], [2, 3], [4, 1]]])
```

Use the `create_using` argument to choose the graph class/type.

6.3.2 networkx.generators.small.LCF_graph

LCF_graph (*n*, *shift_list*, *repeats*, *create_using=None*)

Return the cubic graph specified in LCF notation.

LCF notation (LCF=Lederberg-Coxeter-Fruchte) is a compressed notation used in the generation of various cubic Hamiltonian graphs of high symmetry. See, for example, `dodecahedral_graph`, `desargues_graph`, `heawood_graph` and `pappus_graph` below.

n (number of nodes) The starting graph is the n-cycle with nodes 0,...,n-1. (The null graph is returned if $n < 0$.)

`shift_list = [s1,s2,...,sk]`, a list of integer shifts mod n,

repeats integer specifying the number of times that shifts in `shift_list` are successively applied to each `v_current` in the n-cycle to generate an edge between `v_current` and `v_current+shift mod n`.

For `v1` cycling through the n-cycle a total of `k*repeats` with `shift` cycling through `shiftlist` repeats times connect `v1` with `v1+shift mod n`

The utility graph $K_{\{3,3\}}$

```
>>> G=nx.LCF_graph(6, [3, -3], 3)
```

The Heawood graph

```
>>> G=nx.LCF_graph(14, [5, -5], 7)
```

See <http://mathworld.wolfram.com/LCFNotation.html> for a description and references.

6.3.3 networkx.generators.small.bull_graph

bull_graph (*create_using=None*)

Return the Bull graph.

6.3.4 networkx.generators.small.chvatal_graph

chvatal_graph (*create_using=None*)

Return the Chvatal graph.

6.3.5 networkx.generators.small.cubical_graph

cubical_graph (*create_using=None*)

Return the 3-regular Platonic Cubical graph.

6.3.6 networkx.generators.small.desargues_graph

desargues_graph (*create_using=None*)

Return the Desargues graph.

6.3.7 networkx.generators.small.diamond_graph

diamond_graph (*create_using=None*)

Return the Diamond graph.

6.3.8 networkx.generators.small.dodecahedral_graph

dodecahedral_graph (*create_using=None*)
Return the Platonic Dodecahedral graph.

6.3.9 networkx.generators.small.frucht_graph

frucht_graph (*create_using=None*)
Return the Frucht Graph.

The Frucht Graph is the smallest cubical graph whose automorphism group consists only of the identity element.

6.3.10 networkx.generators.small.heawood_graph

heawood_graph (*create_using=None*)
Return the Heawood graph, a (3,6) cage.

6.3.11 networkx.generators.small.house_graph

house_graph (*create_using=None*)
Return the House graph (square with triangle on top).

6.3.12 networkx.generators.small.house_x_graph

house_x_graph (*create_using=None*)
Return the House graph with a cross inside the house square.

6.3.13 networkx.generators.small.icosahedral_graph

icosahedral_graph (*create_using=None*)
Return the Platonic Icosahedral graph.

6.3.14 networkx.generators.small.krackhardt_kite_graph

krackhardt_kite_graph (*create_using=None*)
Return the Krackhardt Kite Social Network.

A 10 actor social network introduced by David Krackhardt to illustrate: degree, betweenness, centrality, closeness, etc. The traditional labeling is: Andre=1, Beverley=2, Carol=3, Diane=4, Ed=5, Fernando=6, Garth=7, Heather=8, Ike=9, Jane=10.

6.3.15 networkx.generators.small.moebius_kantor_graph

moebius_kantor_graph (*create_using=None*)
Return the Moebius-Kantor graph.

6.3.16 `networkx.generators.small.octahedral_graph`

`octahedral_graph` (*create_using=None*)
Return the Platonic Octahedral graph.

6.3.17 `networkx.generators.small.pappus_graph`

`pappus_graph` ()
Return the Pappus graph.

6.3.18 `networkx.generators.small.petersen_graph`

`petersen_graph` (*create_using=None*)
Return the Petersen graph.

6.3.19 `networkx.generators.small.sedgewick_maze_graph`

`sedgewick_maze_graph` (*create_using=None*)
Return a small maze with a cycle.

This is the maze used in Sedgewick, 3rd Edition, Part 5, Graph Algorithms, Chapter 18, e.g. Figure 18.2 and following. Nodes are numbered 0,...,7

6.3.20 `networkx.generators.small.tetrahedral_graph`

`tetrahedral_graph` (*create_using=None*)
Return the 3-regular Platonic Tetrahedral graph.

6.3.21 `networkx.generators.small.truncated_cube_graph`

`truncated_cube_graph` (*create_using=None*)
Return the skeleton of the truncated cube.

6.3.22 `networkx.generators.small.truncated_tetrahedron_graph`

`truncated_tetrahedron_graph` (*create_using=None*)
Return the skeleton of the truncated Platonic tetrahedron.

6.3.23 `networkx.generators.small.tutte_graph`

`tutte_graph` (*create_using=None*)
Return the Tutte graph.

6.4 Random Graphs

Generators for random graphs.

<code>fast_gnp_random_graph(n, p[, create_using, seed])</code>	Return a random graph $G_{\{n,p\}}$.
<code>gnp_random_graph(n, p[, create_using, seed])</code>	Return a random graph $G_{\{n,p\}}$.
<code>directed_gnp_random_graph(n, p[, ...])</code>	Return a directed random graph.
<code>dense_gnm_random_graph(n, m[, create_using, ...])</code>	Return the random graph $G_{\{n,m\}}$.
<code>gnm_random_graph(n, m[, create_using, seed])</code>	Return the random graph $G_{\{n,m\}}$.
<code>erdos_renyi_graph(n, p[, create_using, seed])</code>	Return a random graph $G_{\{n,p\}}$.
<code>binomial_graph(n, p[, create_using, seed])</code>	Return a random graph $G_{\{n,p\}}$.
<code>newman_watts_strogatz_graph(n, k, p[, ...])</code>	Return a Newman-Watts-Strogatz small world graph.
<code>watts_strogatz_graph(n, k, p[, ...])</code>	Return a Watts-Strogatz small-world graph.
<code>connected_watts_strogatz_graph(n, k, p[, ...])</code>	Return a connected Watts-Strogatz small-world graph.
<code>random_regular_graph(d, n[, create_using, seed])</code>	Return a random regular graph of n nodes each with degree d .
<code>barabasi_albert_graph(n, m[, create_using, seed])</code>	Return random graph using Barabási-Albert preferential attachment model.
<code>powerlaw_cluster_graph(n, m, p[, ...])</code>	Holme and Kim algorithm for growing graphs with powerlaw
<code>random_lobster(n, p1, p2[, create_using, seed])</code>	Return a random lobster.
<code>random_shell_graph(constructor[, ...])</code>	Return a random shell graph for the constructor given.
<code>random_powerlaw_tree(n[, gamma, ...])</code>	Return a tree with a powerlaw degree distribution.
<code>random_powerlaw_tree_sequence(n[, gamma, ...])</code>	Return a degree sequence for a tree with a powerlaw distribution.

6.4.1 networkx.generators.random_graphs.fast_gnp_random_graph

fast_gnp_random_graph ($n, p, create_using=None, seed=None$)

Return a random graph $G_{\{n,p\}}$.

The $G_{\{n,p\}}$ graph chooses each of the possible $[n(n-1)]/2$ edges with probability p .

Sometimes called Erdős-Rényi graph, or binomial graph.

Parameters **n** : int

The number of nodes.

p : float

Probability for edge creation.

create_using : graph, optional (default Graph)

Use specified graph as a container.

seed : int, optional

Seed for random number generator (default=None).

Notes

This algorithm is $O(n+m)$ where m is the expected number of edges $m=p*n*(n-1)/2$.

It should be faster than `gnp_random_graph` when p is small, and the expected number of edges is small, (sparse graph).

References

[R72]

6.4.2 `networkx.generators.random_graphs.gnp_random_graph`

gnp_random_graph (*n*, *p*, *create_using=None*, *seed=None*)

Return a random graph $G_{\{n,p\}}$.

Chooses 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** : int

The number of nodes.

p : float

Probability for edge creation.

create_using : graph, optional (default Graph)

Use specified graph as a container.

seed : int, optional

Seed for random number generator (default=None).

See Also:

`fast_gnp_random_graph`

Notes

This is an $O(n^2)$ algorithm. For sparse graphs (small p) see `fast_gnp_random_graph`.

References

[R73], [R74]

6.4.3 `networkx.generators.random_graphs.directed_gnp_random_graph`

directed_gnp_random_graph (*n*, *p*, *create_using=None*, *seed=None*)

Return a directed random graph.

Chooses each of the possible $n(n-1)$ edges with probability p .

This is a directed version of `G_np`.

Parameters `n` : int

The number of nodes.

`p` : float

Probability for edge creation.

create_using : graph, optional (default DiGraph)

Use specified graph as a container.

seed : int, optional

Seed for random number generator (default=None).

See Also:

`gnp_random_graph`, `fast_gnp_random_graph`

Notes

This is an $O(n^2)$ algorithm.

References

[R68], [R69]

6.4.4 `networkx.generators.random_graphs.dense_gnm_random_graph`

`dense_gnm_random_graph` (*n*, *m*, *create_using=None*, *seed=None*)

Return the random graph $G_{\{n,m\}}$.

Gives a graph picked randomly out of the set of all graphs with *n* nodes and *m* edges. This algorithm should be faster than `gnm_random_graph` for dense graphs.

Parameters `n` : int

The number of nodes.

`m` : int

The number of edges.

create_using : graph, optional (default Graph)

Use specified graph as a container.

seed : int, optional

Seed for random number generator (default=None).

See Also:

`gnm_random_graph`

Notes

Algorithm by Keith M. Briggs Mar 31, 2006. Inspired by Knuth's Algorithm S (Selection sampling technique), in section 3.4.2 of

References

[R67]

6.4.5 `networkx.generators.random_graphs.gnm_random_graph`

gnm_random_graph (*n*, *m*, *create_using=None*, *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** : int

The number of nodes.

m : int

The number of edges.

create_using : graph, optional (default Graph)

Use specified graph as a container.

seed : int, optional

Seed for random number generator (default=None).

6.4.6 `networkx.generators.random_graphs.erdos_renyi_graph`

erdos_renyi_graph (*n*, *p*, *create_using=None*, *seed=None*)

Return a random graph $G_{\{n,p\}}$.

Chooses 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** : int

The number of nodes.

p : float

Probability for edge creation.

create_using : graph, optional (default Graph)

Use specified graph as a container.

seed : int, optional

Seed for random number generator (default=None).

See Also:

`fast_gnp_random_graph`

Notes

This is an $O(n^2)$ algorithm. For sparse graphs (small p) see `fast_gnp_random_graph`.

References

[R70], [R71]

6.4.7 `networkx.generators.random_graphs.binomial_graph`

`binomial_graph` (*n*, *p*, *create_using=None*, *seed=None*)

Return a random graph $G_{\{n,p\}}$.

Chooses 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** : int

The number of nodes.

p : float

Probability for edge creation.

create_using : graph, optional (default Graph)

Use specified graph as a container.

seed : int, optional

Seed for random number generator (default=None).

See Also:

`fast_gnp_random_graph`

Notes

This is an $O(n^2)$ algorithm. For sparse graphs (small p) see `fast_gnp_random_graph`.

References

[R65], [R66]

6.4.8 `networkx.generators.random_graphs.newman_watts_strogatz_graph`

`newman_watts_strogatz_graph` (*n*, *k*, *p*, *create_using=None*, *seed=None*)

Return a Newman-Watts-Strogatz small world graph.

Parameters **n** : int

The number of nodes

k : int

Each node is connected to k nearest neighbors in ring topology

p : float

The probability of adding a new edge for each edge

create_using : graph, optional (default Graph)

The graph instance used to build the graph.

seed : int, optional

seed for random number generator (default=None)

See Also:

`watts_strogatz_graph`

Notes

First create a ring over n nodes. Then each node in the ring is connected with its k nearest neighbors (k-1 neighbors if k is odd). Then shortcuts are created by adding new edges as follows: for each edge u-v in the underlying “n-ring with k nearest neighbors” with probability p add a new edge u-w with randomly-chosen existing node w. In contrast with `watts_strogatz_graph()`, no edges are removed.

References

[R75]

6.4.9 networkx.generators.random_graphs.watts_strogatz_graph

watts_strogatz_graph (*n, k, p, create_using=None, seed=None*)

Return a Watts-Strogatz small-world graph.

Parameters **n** : int

The number of nodes

k : int

Each node is connected to k nearest neighbors in ring topology

p : float

The probability of rewiring each edge

create_using : graph, optional (default Graph)

The graph instance used to build the graph.

seed : int, optional

Seed for random number generator (default=None)

See Also:

`newman_watts_strogatz_graph`, `connected_watts_strogatz_graph`

Notes

First create a ring over n nodes. Then each node in the ring is connected with its k nearest neighbors ($k-1$ neighbors if k is odd). Then shortcuts are created by replacing some edges as follows: for each edge $u-v$ in the underlying “ n -ring with k nearest neighbors” with probability p replace it with a new edge $u-w$ with uniformly random choice of existing node w .

In contrast with `newman_watts_strogatz_graph()`, the random rewiring does not increase the number of edges. The rewired graph is not guaranteed to be connected as in `connected_watts_strogatz_graph()`.

References

[R79]

6.4.10 `networkx.generators.random_graphs.connected_watts_strogatz_graph`

`connected_watts_strogatz_graph` ($n, k, p, \text{tries}=100, \text{create_using}=\text{None}, \text{seed}=\text{None}$)

Return a connected Watts-Strogatz small-world graph.

Attempt to generate a connected realization by repeated generation of Watts-Strogatz small-world graphs. An exception is raised if the maximum number of tries is exceeded.

Parameters n : int

The number of nodes

k : int

Each node is connected to k nearest neighbors in ring topology

p : float

The probability of rewiring each edge

tries : int

Number of attempts to generate a connected graph.

create_using : graph, optional (default Graph)

The graph instance used to build the graph.

seed : int, optional

The seed for random number generator.

See Also:

`newman_watts_strogatz_graph`, `watts_strogatz_graph`

6.4.11 `networkx.generators.random_graphs.random_regular_graph`

`random_regular_graph` ($d, n, \text{create_using}=\text{None}, \text{seed}=\text{None}$)

Return a random regular graph of n nodes each with degree d .

The resulting graph G has no self-loops or parallel edges.

Parameters d : int

Degree

n : integer
 Number of nodes. The value of $n*d$ must be even.

create_using : graph, optional (default Graph)
 The graph instance used to build the graph.

seed : hashable object
 The seed for random number generator.

Notes

The nodes are numbered from 0 to $n-1$.

Kim and Vu's paper [R78] shows that this algorithm samples in an asymptotically uniform way from the space of random graphs when $d = O(n^{1/3-\epsilon})$.

References

[R77], [R78]

6.4.12 networkx.generators.random_graphs.barabasi_albert_graph

barabasi_albert_graph (*n*, *m*, *create_using=None*, *seed=None*)

Return random graph using Barabási-Albert preferential attachment model.

A graph of n nodes is grown by attaching new nodes each with m edges that are preferentially attached to existing nodes with high degree.

Parameters **n** : int

Number of nodes

m : int

Number of edges to attach from a new node to existing nodes

create_using : graph, optional (default Graph)

The graph instance used to build the graph.

seed : int, optional

Seed for random number generator (default=None).

Returns **G** : Graph

Notes

The initialization is a graph with m nodes and no edges.

References

[R64]

6.4.13 networkx.generators.random_graphs.powerlaw_cluster_graph

powerlaw_cluster_graph (*n, m, p, create_using=None, seed=None*)

Holme and Kim algorithm for growing graphs with powerlaw degree distribution and approximate average clustering.

Parameters **n** : int

the number of nodes

m : int

the number of random edges to add for each new node

p : float,

Probability of adding a triangle after adding a random edge

create_using : graph, optional (default Graph)

The graph instance used to build the graph.

seed : int, optional

Seed for random number generator (default=None).

Notes

The average clustering has a hard time getting above a certain cutoff that depends on *m*. This cutoff is often quite low. Note that the transitivity (fraction of triangles to possible triangles) seems to go down with network size.

It is essentially the Barabási-Albert (B-A) growth model with an extra step that each random edge is followed by a chance of making an edge to one of its neighbors too (and thus a triangle).

This algorithm improves on B-A in the sense that it enables a higher average clustering to be attained if desired.

It seems possible to have a disconnected graph with this algorithm since the initial *m* nodes may not be all linked to a new node on the first iteration like the B-A model.

References

[R76]

6.4.14 networkx.generators.random_graphs.random_lobster

random_lobster (*n, p1, p2, create_using=None, seed=None*)

Return a random lobster.

A lobster is a tree that reduces to a caterpillar when pruning all leaf nodes.

A caterpillar is a tree that reduces to a path graph when pruning all leaf nodes (*p2=0*).

Parameters **n** : int

The expected number of nodes in the backbone

p1 : float

Probability of adding an edge to the backbone

p2 : float

Probability of adding an edge one level beyond backbone

create_using : graph, optional (default Graph)

The graph instance used to build the graph.

seed : int, optional

Seed for random number generator (default=None).

6.4.15 networkx.generators.random_graphs.random_shell_graph

random_shell_graph (*constructor, create_using=None, seed=None*)

Return a random shell graph for the constructor given.

Parameters **constructor**: a list of three-tuples :

(n,m,d) for each shell starting at the center shell.

n : int

The number of nodes in the shell

m : int

The number of edges in the shell

d : float

The ratio of inter-shell (next) edges to intra-shell edges. d=0 means no intra shell edges, d=1 for the last shell

create_using : graph, optional (default Graph)

The graph instance used to build the graph.

seed : int, optional

Seed for random number generator (default=None).

Examples

```
>>> constructor=[(10,20,0.8),(20,40,0.8)]
>>> G=nx.random_shell_graph(constructor)
```

6.4.16 networkx.generators.random_graphs.random_powerlaw_tree

random_powerlaw_tree (*n, gamma=3, create_using=None, seed=None, tries=100*)

Return a tree with a powerlaw degree distribution.

Parameters **n** : int,

The number of nodes

gamma : float

Exponent of the power-law

create_using : graph, optional (default Graph)

The graph instance used to build the graph.

seed : int, optional

Seed for random number generator (default=None).

tries : int

Number of attempts to adjust sequence to make a tree

Notes

A trial powerlaw degree sequence is chosen and then elements are swapped with new elements from a powerlaw distribution until the sequence makes a tree (#edges=#nodes-1).

6.4.17 networkx.generators.random_graphs.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.

Parameters **n** : int,

The number of nodes

gamma : float

Exponent of the power-law

seed : int, optional

Seed for random number generator (default=None).

tries : int

Number of attempts to adjust sequence to make a tree

Notes

A trial powerlaw degree sequence is chosen and then elements are swapped with new elements from a powerlaw distribution until the sequence makes a tree (#edges=#nodes-1).

6.5 Degree Sequence

Generate graphs with a given degree sequence or expected degree sequence.

<code>configuration_model(deg_sequence[, ...])</code>	Return a random graph with the given degree sequence.
<code>directed_configuration_model(deg_sequences[, ...])</code>	Return a directed_random graph with the given degree sequences.
<code>expected_degree_graph(w[, create_using, seed])</code>	Return a random graph G(w) with expected degrees given by w.
<code>havel_hakimi_graph(deg_sequence[, create_using])</code>	Return a simple graph with given degree sequence, constructed using the
<code>degree_sequence_tree(deg_sequence[, ...])</code>	Make a tree for the given degree sequence.
<code>is_valid_degree_sequence(deg_sequence)</code>	Return True if deg_sequence is a valid sequence of integer degrees
<code>create_degree_sequence(n, **kwds[, ...])</code>	Attempt to create a valid degree sequence of length n using specified function sfunction(n, **kwds).
<code>double_edge_swap(G[, nswap])</code>	Attempt nswap double-edge swaps on the graph G.
<code>connected_double_edge_swap(G[, nswap])</code>	Attempt nswap double-edge swaps on the graph G.
<code>li_smax_graph(degrees_seq[, create_using])</code>	Generates a graph based with a given degree sequence and maximizing the s-metric.

6.5.1 networkx.generators.degree_seq.configuration_model

configuration_model (*deg_sequence*, *create_using=None*, *seed=None*)

Return a random graph with the given degree sequence.

The configuration model generates a random pseudograph (graph with parallel edges and self loops) by randomly assigning edges to match the given degree sequence.

Parameters *deg_sequence* : list of integers

Each list entry corresponds to the degree of a node.

create_using : graph, optional (default MultiGraph)

Return graph of this type. The instance will be cleared.

seed : hashable object, optional

Seed for random number generator.

Returns *G* : MultiGraph

A graph with the specified degree sequence. Nodes are labeled starting at 0 with an index corresponding to the position in *deg_sequence*.

Raises **NetworkXError** :

If the degree sequence does not have an even sum.

See Also:

`is_valid_degree_sequence`

Notes

As described by Newman [R54].

A non-graphical degree sequence (not realizable by some simple graph) is allowed since this function returns graphs with self loops and parallel edges. An exception is raised if the degree sequence does not have an even sum.

This configuration model construction process can lead to duplicate edges and loops. You can remove the self-loops and parallel edges (see below) which will likely result in a graph that doesn't have the exact degree sequence specified. This "finite-size effect" decreases as the size of the graph increases.

References

[R54]

Examples

```
>>> from networkx.utils import powerlaw_sequence
>>> z=nx.create_degree_sequence(100,powerlaw_sequence)
>>> G=nx.configuration_model(z)
```

To remove parallel edges:

```
>>> G=nx.Graph(G)
```

To remove self loops:

```
>>> G.remove_edges_from(G.selfloop_edges())
```

6.5.2 networkx.generators.degree_seq.directed_configuration_model

directed_configuration_model (*in_degree_sequence*, *out_degree_sequence*, *create_using=None*, *seed=None*)

Return a directed_random graph with the given degree sequences.

The configuration model generates a random directed pseudograph (graph with parallel edges and self loops) by randomly assigning edges to match the given degree sequences.

Parameters *in_degree_sequence* : list of integers

Each list entry corresponds to the in-degree of a node.

out_degree_sequence : list of integers

Each list entry corresponds to the out-degree of a node.

create_using : graph, optional (default MultiDiGraph)

Return graph of this type. The instance will be cleared.

seed : hashable object, optional

Seed for random number generator.

Returns *G* : MultiDiGraph

A graph with the specified degree sequences. Nodes are labeled starting at 0 with an index corresponding to the position in *deg_sequence*.

Raises *NetworkXError* :

If the degree sequences do not have the same sum.

See Also:

`configuration_model`

Notes

Algorithm as described by Newman [R56].

A non-graphical degree sequence (not realizable by some simple graph) is allowed since this function returns graphs with self loops and parallel edges. An exception is raised if the degree sequences does not have the same sum.

This configuration model construction process can lead to duplicate edges and loops. You can remove the self-loops and parallel edges (see below) which will likely result in a graph that doesn't have the exact degree sequence specified. This "finite-size effect" decreases as the size of the graph increases.

References

[R56]

Examples

```
>>> D=nx.DiGraph([(0,1),(1,2),(2,3)]) # directed path graph
>>> din=D.in_degree().values()
>>> dout=D.out_degree().values()
>>> din.append(1)
>>> dout[0]=2
>>> D=nx.directed_configuration_model(din,dout)
```

To remove parallel edges:

```
>>> D=nx.DiGraph(D)
```

To remove self loops:

```
>>> D.remove_edges_from(D.selfloop_edges())
```

6.5.3 networkx.generators.degree_seq.expected_degree_graph

expected_degree_graph (*w*, *create_using=None*, *seed=None*)

Return a random graph $G(w)$ with expected degrees given by *w*.

Parameters *w* : list

The list of expected degrees.

create_using : graph, optional (default Graph)

Return graph of this type. The instance will be cleared.

seed : hashable object, optional

The seed for the random number generator.

References

[R57]

Examples

```
>>> z=[10 for i in range(100)]
>>> G=nx.expected_degree_graph(z)
```

6.5.4 networkx.generators.degree_seq.havel_hakimi_graph

havel_hakimi_graph (*deg_sequence*, *create_using=None*)

Return a simple graph with given degree sequence, constructed using the Havel-Hakimi algorithm.

Parameters **deg_sequence**: list of integers :

Each integer corresponds to the degree of a node (need not be sorted).

create_using : graph, optional (default Graph)

Return graph of this type. The instance will be cleared. Multigraphs and directed graphs are not allowed.

Raises **NetworkXException** :

For a non-graphical degree sequence (i.e. one not realizable by some simple graph).

Notes

The Havel-Hakimi algorithm constructs a simple graph by successively connecting the node of highest degree to other nodes of highest degree, resorting remaining nodes by degree, and repeating the process. The resulting graph has a high degree-associativity. Nodes are labeled 1,..., len(deg_sequence), corresponding to their position in deg_sequence.

See Theorem 1.4 in [chartrand-graphs-1996]. This algorithm is also used in the function `is_valid_degree_sequence`.

References

[R58]

6.5.5 networkx.generators.degree_seq.degree_sequence_tree

degree_sequence_tree (*deg_sequence*, *create_using=None*)

Make a tree for the given degree sequence.

A tree has $\#nodes - \#edges = 1$ so the degree sequence must have $len(deg_sequence) - sum(deg_sequence)/2 = 1$

6.5.6 networkx.generators.degree_seq.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 [R59]. This algorithm is also used in `havel_hakimi_graph()`

References

[R59]

6.5.7 networkx.generators.degree_seq.create_degree_sequence

create_degree_sequence (*n*, *sfunction=None*, *max_tries=50*, ***kws*)

Attempt to create a valid degree sequence of length *n* using specified function `sfunction(n,**kws)`.

Parameters *n* : int

Length of degree sequence = number of nodes

sfunction: function :

Function which returns a list of *n* real or integer values. Called as “`sfunction(n,**kws)`”.

max_tries: int :

Max number of attempts at creating valid degree sequence.

Notes

Repeatedly create a degree sequence by calling `sfunction(n,**kws)` until achieving a valid degree sequence. If unsuccessful after `max_tries` attempts, raise an exception.

For examples of `sfunctions` that return sequences of random numbers, see `networkx.Utils`.

Examples

```
>>> from networkx.utils import uniform_sequence
>>> seq=nx.create_degree_sequence(10,uniform_sequence)
```

6.5.8 networkx.generators.degree_seq.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 chosen edges *u-v* and *x-y* and creates the new edges *u-x* and *v-y*:

```

u--v          u  v
      becomes |  |
x--y          x  y
    
```

If either the edge u-x or v-y already exist no swap is performed so the actual count of swapped edges is always $\leq n_{\text{swap}}$

Does not enforce any connectivity constraints.

6.5.9 networkx.generators.degree_seq.connected_double_edge_swap

connected_double_edge_swap (*G*, *nswap=1*)

Attempt *nswap* double-edge swaps on the graph *G*.

Returns the count of successful swaps. Enforces connectivity. The graph *G* is modified in place.

Notes

A double-edge swap removes two randomly chosen edges u-v and x-y and creates the new edges u-x and v-y:

```

u--v          u  v
      becomes |  |
x--y          x  y
    
```

If either the edge u-x or v-y already exist no swap is performed so the actual count of swapped edges is always $\leq n_{\text{swap}}$

The initial graph *G* must be connected and the resulting graph is connected.

References

[R55]

6.5.10 networkx.generators.degree_seq.li_smax_graph

li_smax_graph (*degree_seq*, *create_using=None*)

Generates a graph based with a given degree sequence and maximizing the s-metric. Experimental implementation.

Maximum s-metric 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  $i < j$ ,  $i \leq k < l$  and
       $d_i * d_j \geq d_k * d_l$ 
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 sundsda1@gmail.com if you can provide python code that can guarantee maximality. Several optimizations are included in this code and it may be hard to read. Commented code to come.

A POSSIBLE ALTERNATIVE:

For an ‘unconstrained’ graph, that is one they describe as having the sum of the degree sequence be even (ie all undirected graphs) they present a simpler algorithm. It is as follows

“For each vertex i: if d_i is even then attach $d_i/2$ self-loops; if d_i is odd, then attach $(d_i-1)/2$ self-loops, leaving one available “stub”. Second for all remaining vertices with “stubs” connect them in pairs according to decreasing values of d_i .”[1]

Since this only works for undirected graphs anyway, perhaps this is the better method? Note this also returns a graph with a larger s_metric than the other method, and it seems to have the same degree sequence, though I haven’t tested it extensively.

6.6 Directed

Generators for some directed graphs.

`gn_graph`: growing network `gnc_graph`: growing network with copying `gnr_graph`: growing network with redirection
`scale_free_graph`: scale free directed graph

<code>gn_graph(n[, kernel, create_using, seed])</code>	Return the GN digraph with n nodes.
<code>gnr_graph(n, p[, create_using, seed])</code>	Return the GNR digraph with n nodes and redirection probability p.
<code>gnc_graph(n[, create_using, seed])</code>	Return the GNC digraph with n nodes.
<code>scale_free_graph(n[, alpha, beta, gamma, ...])</code>	Return a scale free directed graph.

6.6.1 networkx.generators.directed.gn_graph

gn_graph (*n*, *kernel=None*, *create_using=None*, *seed=None*)

Return the GN digraph with n nodes.

The GN (growing network) graph is built by adding nodes one at a time with a link to one previously added node. The target node for the link is chosen with probability based on degree. The default attachment kernel is a linear function of degree.

The graph is always a (directed) tree.

Parameters **n** : int

The number of nodes for the generated graph.

kernel : function

The attachment kernel.

create_using : graph, optional (default DiGraph)

Return graph of this type. The instance will be cleared.

seed : hashable object, optional

The seed for the random number generator.

References

[R60]

Examples

```
>>> D=nx.gn_graph(10)      # the GN graph
>>> G=D.to_undirected()   # the undirected version
```

To specify an attachment kernel use the kernel keyword

```
>>> D=nx.gn_graph(10, kernel=lambda x:x**1.5) # A_k=k^1.5
```

6.6.2 networkx.generators.directed.gnr_graph

gnr_graph (*n*, *p*, *create_using=None*, *seed=None*)

Return the GNR digraph with *n* nodes and redirection probability *p*.

The GNR (growing network with redirection) graph is built by adding nodes one at a time with a link to one previously added node. The previous target node is chosen uniformly at random. With probability *p* the link is instead “redirected” to the successor node of the target. The graph is always a (directed) tree.

Parameters **n** : int

The number of nodes for the generated graph.

p : float

The redirection probability.

create_using : graph, optional (default DiGraph)

Return graph of this type. The instance will be cleared.

seed : hashable object, optional

The seed for the random number generator.

References

[R62]

Examples

```
>>> D=nx.gnr_graph(10,0.5) # the GNR graph
>>> G=D.to_undirected() # the undirected version
```

6.6.3 networkx.generators.directed.gnc_graph

gnc_graph (*n*, *create_using=None*, *seed=None*)

Return the GNC digraph with *n* nodes.

The GNC (growing network with copying) graph is built by adding nodes one at a time with a links to one previously added node (chosen uniformly at random) and to all of that node’s successors.

Parameters **n** : int

The number of nodes for the generated graph.

create_using : graph, optional (default DiGraph)

Return graph of this type. The instance will be cleared.

seed : hashable object, optional

The seed for the random number generator.

References

[R61]

6.6.4 networkx.generators.directed.scale_free_graph

scale_free_graph (*n*, *alpha*=0.40999999999999998, *beta*=0.54000000000000004, *gamma*=0.05000000000000003, *delta_in*=0.20000000000000001, *delta_out*=0, *create_using*=None, *seed*=None)

Return a scale free directed graph.

Parameters **n** : integer

Number of nodes in graph

alpha : float

Probability for adding a new node connected to an existing node chosen randomly according to the in-degree distribution.

beta : float

Probability for adding an edge between two existing nodes. One existing node is chosen randomly according to the in-degree distribution and the other chosen randomly according to the out-degree distribution.

gamma : float

Probability for adding a new node connected to an existing node chosen randomly according to the out-degree distribution.

delta_in : float

Bias for choosing nodes from in-degree distribution.

delta_out : float

Bias for choosing nodes from out-degree distribution.

create_using : graph, optional (default MultiDiGraph)

Use this graph instance to start the process (default=3-cycle).

seed : integer, optional

Seed for random number generator

Notes

The sum of alpha, beta, and gamma must be 1.

References

[R63]

Examples

```
>>> G=nx.scale_free_graph(100)
```

6.7 Geometric

Generators for geometric graphs.

<code>random_geometric_graph(n, radius[, ...])</code>	Random geometric graph in the unit cube
---	---

6.7.1 networkx.generators.geometric.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.

6.8 Hybrid

Hybrid

<code>kl_connected_subgraph(G, k, l[, low_memory, ...])</code>	Returns the maximum locally (k,l) connected subgraph of G.
<code>is_kl_connected(G, k, l[, low_memory])</code>	Returns True if G is kl connected

6.8.1 networkx.generators.hybrid.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

6.8.2 networkx.generators.hybrid.is_kl_connected

is_kl_connected (*G, k, l, low_memory=False*)

Returns True if G is kl connected

6.9 Bipartite

Generators and functions for bipartite graphs.

<code>bipartite_configuration_model(aseq, bseq, ...)</code>	Return a random bipartite graph from two given degree sequences.
<code>bipartite_havel_hakimi_graph(aseq, bseq, ...)</code>	Return a bipartite graph from two given degree sequences
<code>bipartite_reverse_havel_hakimi_graph(aseq, bseq)</code>	Return a bipartite graph from two given degree sequences
<code>bipartite_alternating_havel_hakimi_graph(aseq, bseq)</code>	Return a bipartite graph from two given degree sequences
<code>bipartite_preferential_attachment_graph(p)</code>	Create a bipartite graph with a preferential attachment model from a given single degree sequence.
<code>bipartite_random_regular_graph(d, n[, ...])</code>	UNTESTED: Generate a random bipartite graph.

6.9.1 networkx.generators.bipartite.bipartite_configuration_model

bipartite_configuration_model (*aseq, bseq, create_using=None, seed=None*)
 Return a random bipartite graph from two given degree sequences.

Parameters *aseq* : list or iterator

Degree sequence for node set A.

bseq : list or iterator

Degree sequence for node set B.

create_using : NetworkX graph instance, optional

Return graph of this type.

seed : integer, optional

Seed for random number generator.

Nodes from the set A are connected to nodes in the set B by :

choosing randomly from the possible free stubs, one in A and :

one in B. :

Notes

The sum of the two sequences must be equal: $\text{sum}(aseq)=\text{sum}(bseq)$ If no graph type is specified use `MultiGraph` with parallel edges. If you want a graph with no parallel edges use `create_using=Graph()` but then the resulting degree sequences might not be exact.

6.9.2 networkx.generators.bipartite.bipartite_havel_hakimi_graph

bipartite_havel_hakimi_graph (*aseq, bseq, create_using=None*)
 Return a bipartite graph from two given degree sequences using a Havel-Hakimi style construction.

Parameters *aseq* : list or iterator

Degree sequence for node set A.

bseq : list or iterator

Degree sequence for node set B.

create_using : NetworkX graph instance, optional

Return graph of this type.

Nodes from the set A are connected to nodes in the set B by :
connecting the highest degree nodes in set A to :
the highest degree nodes in set B until all stubs are connected. :

Notes

The sum of the two sequences must be equal: $\text{sum}(\text{aseq})=\text{sum}(\text{bseq})$ If no graph type is specified use MultiGraph with parallel edges. If you want a graph with no parallel edges use `create_using=Graph()` but then the resulting degree sequences might not be exact.

6.9.3 networkx.generators.bipartite.bipartite_reverse_havel_hakimi_graph

bipartite_reverse_havel_hakimi_graph (*aseq, bseq, create_using=None*)

Return a bipartite graph from two given degree sequences using a Havel-Hakimi style construction.

Parameters *aseq* : list or iterator

Degree sequence for node set A.

bseq : list or iterator

Degree sequence for node set B.

create_using : NetworkX graph instance, optional

Return graph of this type.

Nodes from the set A are connected to nodes in the set B by :
connecting the highest degree nodes in set A to :
the lowest degree nodes in set B until all stubs are connected. :

Notes

The sum of the two sequences must be equal: $\text{sum}(\text{aseq})=\text{sum}(\text{bseq})$ If no graph type is specified use MultiGraph with parallel edges. If you want a graph with no parallel edges use `create_using=Graph()` but then the resulting degree sequences might not be exact.

6.9.4 networkx.generators.bipartite.bipartite_alternating_havel_hakimi_graph

bipartite_alternating_havel_hakimi_graph (*aseq, bseq, create_using=None*)

Return a bipartite graph from two given degree sequences using an alternating Havel-Hakimi style construction.

Parameters *aseq* : list or iterator

Degree sequence for node set A.

bseq : list or iterator

Degree sequence for node set B.

create_using : NetworkX graph instance, optional

Return graph of this type.

Nodes from the set A are connected to nodes in the set B by :
connecting the highest degree nodes in set A to :
alternatively the highest and the lowest degree nodes in set :
B until all stubs are connected. :

Notes

The sum of the two sequences must be equal: $\text{sum}(\text{aseq})=\text{sum}(\text{bseq})$ If no graph type is specified use `MultiGraph` with parallel edges. If you want a graph with no parallel edges use `create_using=Graph()` but then the resulting degree sequences might not be exact.

6.9.5 `networkx.generators.bipartite.bipartite_preferential_attachment_graph`

`bipartite_preferential_attachment_graph` (*aseq*, *p*, *create_using=None*, *seed=None*)

Create a bipartite graph with a preferential attachment model from a given single degree sequence.

Parameters *aseq* : list or iterator

Degree sequence for node set A.

p : float

Probability that a new bottom node is added.

create_using : NetworkX graph instance, optional

Return graph of this type.

seed : integer, optional

Seed for random number generator.

Notes

@[article{guillaume-2004-bipartite](#), author = {Jean-Loup Guillaume and Matthieu Latapy}, title = {Bipartite structure of all complex networks}, journal = {Inf. Process. Lett.}, volume = {90}, number = {5}, year = {2004}, issn = {0020-0190}, pages = {215–221}, doi = {<http://dx.doi.org/10.1016/j.ipl.2004.03.007>}, publisher = {Elsevier North-Holland, Inc.}, address = {Amsterdam, The Netherlands, The Netherlands}, }

6.9.6 `networkx.generators.bipartite.bipartite_random_regular_graph`

`bipartite_random_regular_graph` (*d*, *n*, *create_using=None*, *seed=None*)

UNTESTED: Generate a random bipartite graph.

Parameters **d** : integer

Degree of graph.

n : integer

Number of nodes in graph.

create_using : NetworkX graph instance, optional

Return graph of this type.

seed : integer, optional

Seed for random number generator.

Notes

Nodes are numbered 0...n-1.

Restrictions on n and d:

- n must be even
- $n \geq 2*d$

Algorithm inspired by `random_regular_graph()`

6.10 Line Graph

Line graphs.

`line_graph(G)` Return the line graph of the graph or digraph G.

6.10.1 networkx.generators.line.line_graph

`line_graph(G)`

Return the line graph of the graph or digraph G.

The line graph of a graph G has a node for each edge in G and an edge between those nodes if the two edges in G share a common node.

For DiGraphs an edge an edge represents a directed path of length 2.

The original node labels are kept as two-tuple node labels in the line graph.

Parameters **G** : graph

A NetworkX Graph or DiGraph

Notes

Not implemented for MultiGraph or MultiDiGraph classes.

Graph, node, and edge data are not propagated to the new graph.

Examples

```
>>> G=nx.star_graph(3)
>>> L=nx.line_graph(G)
>>> print sorted(L.edges()) # makes a clique, K3
[((0, 1), (0, 2)), ((0, 1), (0, 3)), ((0, 3), (0, 2))]
```

6.11 Ego Graph

Ego graph.

`ego_graph(G, n[, radius, center, undirected])` Returns induced subgraph of neighbors centered at node n.

6.11.1 networkx.generators.ego.ego_graph

ego_graph (*G*, *n*, *radius=1*, *center=True*, *undirected=False*)
Returns induced subgraph of neighbors centered at node n.

Parameters **G** : graph

A NetworkX Graph or DiGraph

n : node

A single node

radius : integer, optional

Include all neighbors of distance \leq radius from n

center : bool, optional

If False, do not include center node in graph

undirected: bool, optional :

If True use both in- and out-neighbors of directed graphs.

Notes

For directed graphs *D* this produces the “out” neighborhood or successors. If you want the neighborhood of predecessors first reverse the graph with *D.reverse()*. If you want both directions use the keyword argument `undirected=True`.

6.12 Stochastic

Stochastic graph.

`stochastic_graph(G[, copy])` Return a right-stochastic representation of *G*.

6.12.1 networkx.generators.stochastic.stochastic_graph

stochastic_graph (*G*, *copy=True*)
Return a right-stochastic representation of *G*.

A right-stochastic graph is a weighted graph in which all of the node (out) neighbors edge weights sum to 1.

Parameters **G** : graph

A NetworkX graph, must have valid edge weights

copy : boolean, optional

If True make a copy of the graph, otherwise modify original graph

LINEAR ALGEBRA

7.1 Spectrum

Laplacian, adjacency matrix, and spectrum of graphs.

<code>adj_matrix(G[, nodelist])</code>	Return adjacency matrix of G.
<code>laplacian(G[, nodelist])</code>	Return the Laplacian matrix of G.
<code>normalized_laplacian(G[, nodelist])</code>	Return the normalized Laplacian matrix of G.
<code>laplacian_spectrum(G)</code>	Return eigenvalues of the Laplacian of G
<code>adjacency_spectrum(G)</code>	Return eigenvalues of the adjacency matrix of G.

7.1.1 networkx.linalg.spectrum.adj_matrix

adj_matrix(*G*, *nodelist=None*)

Return adjacency matrix of G.

Parameters **G** : graph

A NetworkX graph

nodelist : list, optional

The rows and columns are ordered according to the nodes in nodelist. If nodelist is None, then the ordering is produced by G.nodes().

Returns **A** : numpy matrix

Adjacency matrix representation of G.

See Also:

`to_numpy_matrix`, `to_dict_of_dicts`

Notes

If you want a pure Python adjacency matrix representation try `networkx.convert.to_dict_of_dicts` which will return a dictionary-of-dictionaries format that can be addressed as a sparse matrix.

7.1.2 networkx.linalg.spectrum.laplacian

laplacian (*G*, *nodelist=None*)

Return the Laplacian matrix of *G*.

The graph Laplacian is the matrix $L = D - A$, where *A* is the adjacency matrix and *D* is the diagonal matrix of node degrees.

Parameters *G* : graph

A NetworkX graph

nodelist : list, optional

The rows and columns are ordered according to the nodes in *nodelist*. If *nodelist* is *None*, then the ordering is produced by *G.nodes()*.

Returns *L* : NumPy matrix

Laplacian of *G*.

See Also:

`normalized_laplacian`

7.1.3 networkx.linalg.spectrum.normalized_laplacian

normalized_laplacian (*G*, *nodelist=None*)

Return the normalized Laplacian matrix of *G*.

The normalized graph Laplacian is the matrix $NL = D^{-1/2} L D^{-1/2}$ *L* is the graph Laplacian and *D* is the diagonal matrix of node degrees.

Parameters *G* : graph

A NetworkX graph

nodelist : list, optional

The rows and columns are ordered according to the nodes in *nodelist*. If *nodelist* is *None*, then the ordering is produced by *G.nodes()*.

Returns *L* : NumPy matrix

Normalized Laplacian of *G*.

See Also:

`laplacian`

References

[R86]

7.1.4 networkx.linalg.spectrum.laplacian_spectrum

laplacian_spectrum (*G*)

Return eigenvalues of the Laplacian of *G*

Parameters *G* : graph

A NetworkX graph

Returns `evals` : NumPy array

Eigenvalues

See Also:

`laplacian`

7.1.5 `networkx.linalg.spectrum.adjacency_spectrum`

`adjacency_spectrum`(*G*)

Return eigenvalues of the adjacency matrix of *G*.

Parameters *G* : graph

A NetworkX graph

Returns `evals` : NumPy array

Eigenvalues

See Also:

`adj_matrix`

7.2 Attribute Matrices

Functions for constructing matrix-like objects from graph attributes.

<code>attr_matrix</code> (<i>G</i> [, <i>edge_attr</i> , <i>node_attr</i> , ...])	Returns a NumPy matrix using attributes from <i>G</i> .
<code>attr_sparse_matrix</code> (<i>G</i> [, <i>edge_attr</i> , ...])	Returns a SciPy sparse matrix using attributes from <i>G</i> .

7.2.1 `networkx.linalg.attrmatrix.attr_matrix`

`attr_matrix`(*G*, *edge_attr*=None, *node_attr*=None, *normalized*=False, *rc_order*=None, *dtype*=None, *order*=None)

Returns a NumPy matrix using attributes from *G*.

If only *G* is passed in, then the adjacency matrix is constructed.

Let *A* be a discrete set of values for the node attribute *node_attr*. Then the elements of *A* represent the rows and columns of the constructed matrix. Now, iterate through every edge *e*=(*u*,*v*) in *G* and consider the value of the edge attribute *edge_attr*. If *ua* and *va* are the values of the node attribute *node_attr* for *u* and *v*, respectively, then the value of the edge attribute is added to the matrix element at (*ua*, *va*).

Parameters *G* : graph

The NetworkX graph used to construct the NumPy matrix.

`edge_attr` : str, optional

Each element of the matrix represents a running total of the specified edge attribute for edges whose node attributes correspond to the rows/cols of the matrix. The attribute must be present for all edges in the graph. If no attribute is specified, then we just count the number of edges whose node attributes correspond to the matrix element.

`node_attr` : str, optional

Each row and column in the matrix represents a particular value of the node attribute. The attribute must be present for all nodes in the graph. Note, the values of this attribute should be reliably hashable. So, float values are not recommended. If no attribute is specified, then the rows and columns will be the nodes of the graph.

normalized : bool, optional

If True, then each row is normalized by the summation of its values.

rc_order : list, optional

A list of the node attribute values. This list specifies the ordering of rows and columns of the array. If no ordering is provided, then the ordering will be random (and also, a return value).

Returns **M** : NumPy matrix

The attribute matrix.

ordering : list

If *rc_order* was specified, then only the matrix is returned. However, if *rc_order* was None, then the ordering used to construct the matrix is returned as well.

Examples

Construct an adjacency matrix:

```
>>> G = nx.Graph()
>>> G.add_edge(0,1,thickness=1,weight=3)
>>> G.add_edge(0,2,thickness=2)
>>> G.add_edge(1,2,thickness=3)
>>> nx.attr_matrix(G, rc_order=[0,1,2])
matrix([[ 0.,  1.,  1.],
        [ 1.,  0.,  1.],
        [ 1.,  1.,  0.]])
```

Alternatively, we can obtain the matrix describing edge thickness.

```
>>> nx.attr_matrix(G, edge_attr='thickness', rc_order=[0,1,2])
matrix([[ 0.,  1.,  2.],
        [ 1.,  0.,  3.],
        [ 2.,  3.,  0.]])
```

We can also color the nodes and ask for the probability distribution over all edges (u,v) describing:

$\Pr(v \text{ has color } Y \mid u \text{ has color } X)$

```
>>> G.node[0]['color'] = 'red'
>>> G.node[1]['color'] = 'red'
>>> G.node[2]['color'] = 'blue'
>>> rc = ['red', 'blue']
>>> nx.attr_matrix(G, node_attr='color', normalized=True, rc_order=rc)
matrix([[ 0.33333333,  0.66666667],
        [ 1.,          ,  0.          ]])
```

For example, the above tells us that for all edges (u,v):

$$\Pr(v \text{ is red} \mid u \text{ is red}) = 1/3 \quad \Pr(v \text{ is blue} \mid u \text{ is red}) = 2/3$$

$$\Pr(v \text{ is red} \mid u \text{ is blue}) = 1 \quad \Pr(v \text{ is blue} \mid u \text{ is blue}) = 0$$

Finally, we can obtain the total weights listed by the node colors.

```
>>> nx.attr_matrix(G, edge_attr='weight', node_attr='color', rc_order=rc)
matrix([[ 3.,  2.],
        [ 2.,  0.]])
```

Thus, the total weight over all edges (u,v) with u and v having colors:

(red, red) is 3 # the sole contribution is from edge (0,1) (red, blue) is 2 # contributions from edges (0,2) and (1,2) (blue, red) is 2 # same as (red, blue) since graph is undirected (blue, blue) is 0 # there are no edges with blue endpoints

7.2.2 networkx.linalg.attrmatrix.attr_sparse_matrix

attr_sparse_matrix (*G*, *edge_attr=None*, *node_attr=None*, *normalized=False*, *rc_order=None*, *dtype=None*)

Returns a SciPy sparse matrix using attributes from *G*.

If only *G* is passed in, then the adjacency matrix is constructed.

Let *A* be a discrete set of values for the node attribute *node_attr*. Then the elements of *A* represent the rows and columns of the constructed matrix. Now, iterate through every edge *e*=(u,v) in *G* and consider the value of the edge attribute *edge_attr*. If *ua* and *va* are the values of the node attribute *node_attr* for *u* and *v*, respectively, then the value of the edge attribute is added to the matrix element at (*ua*, *va*).

Parameters **G** : graph

The NetworkX graph used to construct the NumPy matrix.

edge_attr : str, optional

Each element of the matrix represents a running total of the specified edge attribute for edges whose node attributes correspond to the rows/cols of the matrix. The attribute must be present for all edges in the graph. If no attribute is specified, then we just count the number of edges whose node attributes correspond to the matrix element.

node_attr : str, optional

Each row and column in the matrix represents a particular value of the node attribute. The attribute must be present for all nodes in the graph. Note, the values of this attribute should be reliably hashable. So, float values are not recommended. If no attribute is specified, then the rows and columns will be the nodes of the graph.

normalized : bool, optional

If True, then each row is normalized by the summation of its values.

rc_order : list, optional

A list of the node attribute values. This list specifies the ordering of rows and columns of the array. If no ordering is provided, then the ordering will be random (and also, a return value).

Returns **M** : SciPy sparse matrix

The attribute matrix.

ordering : list

If `rc_order` was specified, then only the matrix is returned. However, if `rc_order` was `None`, then the ordering used to construct the matrix is returned as well.

Examples

Construct an adjacency matrix:

```
>>> G = nx.Graph()
>>> G.add_edge(0,1,thickness=1,weight=3)
>>> G.add_edge(0,2,thickness=2)
>>> G.add_edge(1,2,thickness=3)
>>> M = nx.attr_sparse_matrix(G, rc_order=[0,1,2])
>>> M.todense()
matrix([[ 0.,  1.,  1.],
        [ 1.,  0.,  1.],
        [ 1.,  1.,  0.]])
```

Alternatively, we can obtain the matrix describing edge thickness.

```
>>> M = nx.attr_sparse_matrix(G, edge_attr='thickness', rc_order=[0,1,2])
>>> M.todense()
matrix([[ 0.,  1.,  2.],
        [ 1.,  0.,  3.],
        [ 2.,  3.,  0.]])
```

We can also color the nodes and ask for the probability distribution over all edges (u,v) describing:

$\Pr(v \text{ has color } Y \mid u \text{ has color } X)$

```
>>> G.node[0]['color'] = 'red'
>>> G.node[1]['color'] = 'red'
>>> G.node[2]['color'] = 'blue'
>>> rc = ['red', 'blue']
>>> M = nx.attr_sparse_matrix(G, node_attr='color',
>>> M.todense()
matrix([[ 0.33333333,  0.66666667],
        [ 1.          ,  0.          ]])
```

normalized

For example, the above tells us that for all edges (u,v):

$\Pr(v \text{ is red} \mid u \text{ is red}) = 1/3$ $\Pr(v \text{ is blue} \mid u \text{ is red}) = 2/3$

$\Pr(v \text{ is red} \mid u \text{ is blue}) = 1$ $\Pr(v \text{ is blue} \mid u \text{ is blue}) = 0$

Finally, we can obtain the total weights listed by the node colors.

```
>>> M = nx.attr_sparse_matrix(G, edge_attr='weight',
>>> M.todense()
matrix([[ 3.,  2.],
        [ 2.,  0.]])
```

node_attr=

Thus, the total weight over all edges (u,v) with u and v having colors:

(red, red) is 3 # the sole contribution is from edge (0,1) (red, blue) is 2 # contributions from edges (0,2) and (1,2) (blue, red) is 2 # same as (red, blue) since graph is undirected (blue, blue) is 0 # there are no edges with blue endpoints

CONVERTING TO AND FROM OTHER DATA FORMATS

8.1 To NetworkX Graph

This module provides functions to convert NetworkX graphs to and from other formats.

The preferred way of converting data to a NetworkX graph is through the graph constructor. The constructor calls the `to_networkx_graph()` function which attempts to guess the input type and convert it automatically.

8.1.1 Examples

Create a 10 node random graph from a numpy matrix

```
>>> import numpy
>>> a=numpy.reshape(numpy.random.random_integers(0,1,size=100),(10,10))
>>> D=nx.DiGraph(a)
```

or equivalently

```
>>> D=nx.to_networkx_graph(a,create_using=nx.DiGraph())
```

Create a graph with a single edge from a dictionary of dictionaries

```
>>> d={0: {1: 1}} # dict-of-dicts single edge (0,1)
>>> G=nx.Graph(d)
```

8.1.2 See Also

`nx_pygraphviz`, `nx_pydot`

`to_networkx_graph(data[, create_using, ...])` Make a NetworkX graph from a known data structure.

8.1.3 `networkx.convert.to_networkx_graph`

`to_networkx_graph(data, create_using=None, multigraph_input=False)`

Make a NetworkX graph from a known data structure.

The preferred way to call this is automatically from the class constructor

```
>>> d={0: {1: {'weight':1}}} # dict-of-dicts single edge (0,1)
>>> G=nx.Graph(d)
```

instead of the equivalent

```
>>> G=nx.from_dict_of_dicts(d)
```

Parameters `data` : a object to be converted

Current known types are: any NetworkX graph dict-of-dicts dist-of-lists list of edges
numpy matrix numpy ndarray scipy sparse matrix pygraphviz agraph

create_using : NetworkX graph

Use specified graph for result. Otherwise a new graph is created.

multigraph_input : bool (default False)

If True and data is a dict_of_dicts, try to create a multigraph assuming dict_of_dict_of_lists. If data and create_using are both multigraphs then create a multigraph from a multigraph.

8.2 Relabeling

<code>convert_node_labels_to_integers(G[, ...])</code>	Return a copy of G node labels replaced with integers.
<code>relabel_nodes(G, mapping)</code>	Return a copy of G with node labels transformed by mapping.

8.2.1 networkx.convert.convert_node_labels_to_integers

convert_node_labels_to_integers (*G*, *first_label*=0, *ordering*='default', *discard_old_labels*=True)
Return a copy of G node labels replaced with integers.

Parameters `G` : graph

A NetworkX graph

first_label : int, optional (default=0)

An integer specifying the offset in numbering nodes. The n new integer labels are numbered first_label, ..., n+first_label.

ordering : string

“default” : inherit node ordering from G.nodes() “sorted” : inherit node ordering from sorted(G.nodes()) “increasing degree” : nodes are sorted by increasing degree “decreasing degree” : nodes are sorted by decreasing degree

discard_old_labels : bool, optional (default=True)

if True (default) discard old labels if False, create a dict self.node_labels that maps new labels to old labels

8.2.2 networkx.convert.relabel_nodes

relabel_nodes (*G*, *mapping*)

Return a copy of *G* with node labels transformed by mapping.

Parameters **G** : graph

A NetworkX graph

mapping : dictionary or function

Either a dictionary with the old labels as keys and new labels as values or a function transforming an old label with a new label. In either case, the new labels must be hashable Python objects.

See Also:

`convert_node_labels_to_integers`

Examples

mapping as dictionary

```
>>> G=nx.path_graph(3) # nodes 0-1-2
>>> mapping={0:'a',1:'b',2:'c'}
>>> H=nx.relabel_nodes(G,mapping)
>>> print H.nodes()
['a', 'c', 'b']

>>> G=nx.path_graph(26) # nodes 0..25
>>> mapping=dict(zip(G.nodes(),"abcdefghijklmnopqrstuvwxy"))
>>> H=nx.relabel_nodes(G,mapping) # nodes a..z
>>> mapping=dict(zip(G.nodes(),xrange(1,27)))
>>> G1=nx.relabel_nodes(G,mapping) # nodes 1..26
```

mapping as function

```
>>> G=nx.path_graph(3)
>>> def mapping(x):
...     return x**2
>>> H=nx.relabel_nodes(G,mapping)
>>> print H.nodes()
[0, 1, 4]
```

8.3 Dictionaries

<code>to_dict_of_dicts(G[, nodelist, edge_data])</code>	Return adjacency representation of graph as a dictionary of dictionaries
<code>from_dict_of_dicts(d[, create_using, ...])</code>	Return a graph from a dictionary of dictionaries.

8.3.1 `networkx.convert.to_dict_of_dicts`

`to_dict_of_dicts` (*G*, *nodelist=None*, *edge_data=None*)

Return adjacency representation of graph as a dictionary of dictionaries

Parameters *G* : graph

A NetworkX graph

nodelist : list

Use only nodes specified in nodelist

edge_data : list, optional

If provided, the value of the dictionary will be set to *edge_data* for all edges. This is useful to make an adjacency matrix type representation with 1 as the edge data. If *edge_data* is None, the *edge_data* in *G* is used to fill the values. If *G* is a multigraph, the *edge_data* is a dict for each pair (u,v).

8.3.2 `networkx.convert.from_dict_of_dicts`

`from_dict_of_dicts` (*d*, *create_using=None*, *multigraph_input=False*)

Return a graph from a dictionary of dictionaries.

Parameters *d* : dictionary of dictionaries

A dictionary of dictionaries adjacency representation.

create_using : NetworkX graph

Use specified graph for result. Otherwise a new graph is created.

multigraph_input : bool (default False)

When True, the values of the inner dict are assumed to be containers of edge data for multiple edges. Otherwise this routine assumes the edge data are singletons.

Examples

```
>>> dod= {0: {1: {'weight':1}}} # single edge (0,1)
>>> G=nx.from_dict_of_dicts(dod)
```

```
or >>> G=nx.Graph(dod) # use Graph constructor
```

8.4 Lists

<code>to_dict_of_lists</code> (<i>G</i> , <i>nodelist</i>)	Return adjacency representation of graph as a dictionary of lists
<code>from_dict_of_lists</code> (<i>d</i> [, <i>create_using</i>])	Return a graph from a dictionary of lists.
<code>to_edgelist</code> (<i>G</i> , <i>nodelist</i>)	Return a list of edges in the graph.
<code>from_edgelist</code> (<i>edgelist</i> [, <i>create_using</i>])	Return a graph from a list of edges.

8.4.1 `networkx.convert.to_dict_of_lists`

`to_dict_of_lists` (*G*, *nodelist=None*)

Return adjacency representation of graph as a dictionary of lists

Parameters *G* : graph

A NetworkX graph

nodelist : list

Use only nodes specified in nodelist

Notes

Completely ignores edge data for MultiGraph and MultiDiGraph.

8.4.2 `networkx.convert.from_dict_of_lists`

`from_dict_of_lists` (*d*, *create_using=None*)

Return a graph from a dictionary of lists.

Parameters *d* : dictionary of lists

A dictionary of lists adjacency representation.

create_using : NetworkX graph

Use specified graph for result. Otherwise a new graph is created.

Examples

```
>>> dol= {0:[1]} # single edge (0,1)
```

```
>>> G=nx.from_dict_of_lists(dol)
```

or >>> G=nx.Graph(dol) # use Graph constructor

8.4.3 `networkx.convert.to_edgelist`

`to_edgelist` (*G*, *nodelist=None*)

Return a list of edges in the graph.

Parameters *G* : graph

A NetworkX graph

nodelist : list

Use only nodes specified in nodelist

8.4.4 networkx.convert.from_edgelist

from_edgelist (*edgelist*, *create_using=None*)

Return a graph from a list of edges.

Parameters **edgelist** : list or iterator

Edge tuples

create_using : NetworkX graph

Use specified graph for result. Otherwise a new graph is created.

Examples

```
>>> edgelist= [(0,1)] # single edge (0,1)
>>> G=nx.from_edgelist(edgelist)
```

or >>> G=nx.Graph(edgelist) # use Graph constructor

8.5 Numpy

<code>to_numpy_matrix(G[, nodelist, dtype, order])</code>	Return the graph adjacency matrix as a NumPy matrix.
<code>from_numpy_matrix(A[, create_using])</code>	Return a graph from numpy matrix adjacency list.

8.5.1 networkx.convert.to_numpy_matrix

to_numpy_matrix (*G*, *nodelist=None*, *dtype=None*, *order=None*)

Return the graph adjacency matrix as a NumPy matrix.

Parameters **G** : graph

The NetworkX graph used to construct the NumPy matrix.

nodelist : list, optional

The rows and columns are ordered according to the nodes in *nodelist*. If *nodelist* is None, then the ordering is produced by `G.nodes()`.

dtype : NumPy data-type, optional

A valid NumPy dtype used to initialize the array. If None, then the NumPy default is used.

order : {'C', 'F'}, optional

Whether to store multidimensional data in C- or Fortran-contiguous (row- or column-wise) order in memory. If None, then the NumPy default is used.

Returns **M** : NumPy matrix

Graph adjacency matrix.

Notes

The matrix entries are populated using the ‘weight’ edge attribute. When an edge does not have the ‘weight’ attribute, the value of the entry is 1. For multiple edges, the values of the entries are the sums of the edge attributes for each edge.

When *nodelist* does not contain every node in *G*, the matrix is built from the subgraph of *G* that is induced by the nodes in *nodelist*.

Examples

```
>>> G = nx.MultiDiGraph()
>>> G.add_edge(0,1,weight=2)
>>> G.add_edge(1,0)
>>> G.add_edge(2,2,weight=3)
>>> G.add_edge(2,2)
>>> nx.to_numpy_matrix(G, nodelist=[0,1,2])
matrix([[ 0.,  2.,  0.],
        [ 1.,  0.,  0.],
        [ 0.,  0.,  4.]])
```

8.5.2 networkx.convert.from_numpy_matrix

from_numpy_matrix(*A*, *create_using=None*)

Return a graph from numpy matrix adjacency list.

Parameters *A* : numpy matrix

An adjacency matrix representation of a graph

create_using : NetworkX graph

Use specified graph for result. The default is Graph()

Examples

```
>>> import numpy
>>> A=numpy.matrix([[1,1],[2,1]])
>>> G=nx.from_numpy_matrix(A)
```

8.6 Scipy

<code>to_scipy_sparse_matrix(G[, nodelist, dtype])</code>	Return the graph adjacency matrix as a SciPy sparse matrix.
<code>from_scipy_sparse_matrix(A[, create_using])</code>	Return a graph from scipy sparse matrix adjacency list.

8.6.1 networkx.convert.to_scipy_sparse_matrix

to_scipy_sparse_matrix(*G*, *nodelist=None*, *dtype=None*)

Return the graph adjacency matrix as a SciPy sparse matrix.

Parameters **G** : graph

The NetworkX graph used to construct the NumPy matrix.

odelist : list, optional

The rows and columns are ordered according to the nodes in *odelist*. If *odelist* is None, then the ordering is produced by `G.nodes()`.

dtype : NumPy data-type, optional

A valid NumPy dtype used to initialize the array. If None, then the NumPy default is used.

Returns **M** : SciPy sparse matrix

Graph adjacency matrix.

Notes

The matrix entries are populated using the ‘weight’ edge attribute. When an edge does not have the ‘weight’ attribute, the value of the entry is 1. For multiple edges, the values of the entries are the sums of the edge attributes for each edge.

When *odelist* does not contain every node in *G*, the matrix is built from the subgraph of *G* that is induced by the nodes in *odelist*.

Uses `lil_matrix` format. To convert to other formats see the documentation for `scipy.sparse`.

Examples

```
>>> G = nx.MultiDiGraph()
>>> G.add_edge(0,1,weight=2)
>>> G.add_edge(1,0)
>>> G.add_edge(2,2,weight=3)
>>> G.add_edge(2,2)
>>> S = nx.to_scipy_sparse_matrix(G, nodelist=[0,1,2])
>>> S.todense()
matrix([[ 0.,  2.,  0.],
        [ 1.,  0.,  0.],
        [ 0.,  0.,  4.]])
```

8.6.2 `networkx.convert_from_scipy_sparse_matrix`

from_scipy_sparse_matrix(*A*, *create_using=None*)

Return a graph from scipy sparse matrix adjacency list.

Parameters **A** : scipy sparse matrix

An adjacency matrix representation of a graph

create_using : NetworkX graph

Use specified graph for result. The default is `Graph()`

Examples

```
>>> import scipy.sparse
>>> A=scipy.sparse.eye(2,2,1)
>>> G=nx.from_scipy_sparse_matrix(A)
```


READING AND WRITING GRAPHS

9.1 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 `read_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)
```

<code>read_adjlist(path[, comments, delimiter, ...])</code>	Read graph in single line adjacency list format from path.
<code>write_adjlist(G, path[, comments, delimiter])</code>	Write graph G in single-line adjacency-list format to path.
<code>read_multiline_adjlist(path[, comments, ...])</code>	Read graph in multi-line adjacency list format from path.
<code>write_multiline_adjlist(G, path[, ...])</code>	Write the graph G in multiline adjacency list format to the file

9.1.1 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
```

9.1.2 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'
```

9.1.3 networkx.read_multiline_adjlist

read_multiline_adjlist (*path*, *comments='#'*, *delimiter=' '*, *create_using=None*, *nodetype=None*, *edgetype=None*)
 Read graph in multi-line adjacency list format from path.

Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_multiline_adjlist(G,"test.adjlist")
>>> G=nx.read_multiline_adjlist("test.adjlist")
```

path can be a filehandle or a string with the name of the file.

```
>>> fh=open("test.adjlist")
>>> G=nx.read_multiline_adjlist(fh)
```

Filenames ending in .gz or .bz2 will be compressed.

```
>>> nx.write_multiline_adjlist(G,"test.adjlist.gz")
>>> G=nx.read_multiline_adjlist("test.adjlist.gz")
```

nodetype is an optional function to convert node strings to nodetype

For example

```
>>> G=nx.read_multiline_adjlist("test.adjlist", nodetype=int)
```

will attempt to convert all nodes to integer type

Since nodes must be hashable, the function nodetype must return hashable types (e.g. int, float, str, frozenset - or tuples of those, etc.)

edgetype is a function to convert edge data strings to edgetype

```
>>> G=nx.read_multiline_adjlist("test.adjlist")
```

create_using is an optional networkx graph type, the default is Graph(), a simple undirected graph

```
>>> G=nx.read_multiline_adjlist("test.adjlist", create_using=nx.DiGraph())
```

The comments character (default='#') at the beginning of a line indicates a comment line.

The entries are separated by delimiter (default=' '). If whitespace is significant in node or edge labels you should use some other delimiter such as a tab or other symbol.

Example multiline adjlist file format

No edge data:

```
# source target for Graph or DiGraph
a 2
b
c
d 1
e
```

With 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)
```

9.1.4 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)
```

9.2 Edge List

Read and write NetworkX graphs as edge lists.

You can read or write three formats of edge lists with these functions.

Node pairs:

```
1 2 # no data
```

Dictionary as data:

```
1 2 {'weight':7, 'color':'green'}
```

Arbitrary data:

```
1 2 7 green
```

See the `read_edgelist()` function for details and examples.

<code>read_edgelist(path[, comments, delimiter, ...])</code>	Read a graph from a list of edges.
<code>write_edgelist(G, path[, comments, ...])</code>	Write graph as a list of edges.
<code>read_weighted_edgelist(path[, comments, ...])</code>	Read list of edges with numeric weights.
<code>write_weighted_edgelist(G, path[, comments, ...])</code>	Write graph G as a list of edges with numeric weights.

9.2.1 networkx.read_edgelist

read_edgelist (*path*, *comments*='#', *delimiter*=' ', *create_using*=None, *nodetype*=None, *data*=True, *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 used to separate values. The default is whitespace.

create_using : Graph container, optional,

Use specified container to build graph. The default is `networkx.Graph`, an undirected graph.

nodetype : int, float, str, Python type, optional

Convert node data from strings to specified type

data : list of (label,type) tuples

Tuples specifying dictionary key names and types for edge data

edgetype : int, float, str, Python type, optional OBSOLETE

Convert edge data from strings to specified type and use as 'weight'

Returns *G* : graph

A networkx Graph or other type specified with create_using

Notes

Since nodes must be hashable, the function nodetype must return hashable types (e.g. int, float, str, frozenset - or tuples of those, etc.)

Example edgelist file formats.

Without edge data:

```
# read with
# >>> G=nx.read_edgelist(fh,data=False)
# source target
a b
a c
d e
```

With edge data as dictionary:

```
# read with
# >>> G=nx.read_edgelist(fh,data=True)
# source target data
a b {'weight': 1}
a c {'weight': 3.14159}
d e {'fruit': 'apple'}
```

With arbitrary edge data:

```
# read with
# >>> G=nx.read_edgelist(fh,data=[('weight', float)])
# or
# >>> G=nx.read_weighted_edgelist(fh)
# source target data
a b 1
a c 3.14159
d e 42
```

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())
```


9.2.2 networkx.write_edgelist

write_edgelist (*G*, *path*, *comments*='#', *delimiter*=' ', *data*=True)

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 used to separate values. The default is whitespace.

data : bool or list, optional

If False write no edge data. If True write a string representation of the edge data dictionary.. If a list (or other iterable) is provided, write the keys specified in the list.

See Also:

`write_edgelist`, `write_weighted_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.

Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_edgelist(G, "test.edgelist")
>>> G=nx.path_graph(4)
>>> fh=open("test.edgelist",'w')
>>> nx.write_edgelist(G, fh)
>>> nx.write_edgelist(G, "test.edgelist.gz")
>>> nx.write_edgelist(G, "test.edgelist.gz", data=False)

>>> import sys
>>> G=nx.Graph()
>>> G.add_edge(1,2,weight=7,color='red')
>>> nx.write_edgelist(G,sys.stdout,data=False)
1 2
>>> nx.write_edgelist(G,sys.stdout,data=['color'])
1 2 red
>>> nx.write_edgelist(G,sys.stdout,data=['color','weight'])
1 2 red 7
```

9.2.3 networkx.read_weighted_edgelist

read_weighted_edgelist (*path*, *comments*='#', *delimiter*=' ', *create_using*=None, *nodetype*=None)
Read list of edges with numeric weights.

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 used to separate values. The default is whitespace.

create_using : Graph container, optional,

Use specified container to build graph. The default is networkx.Graph, an undirected graph.

nodetype : int, float, str, Python type, optional

Convert node data from strings to specified type

Returns **G** : graph

A networkx Graph or other type specified with create_using

Notes

Since nodes must be hashable, the function nodetype must return hashable types (e.g. int, float, str, frozenset - or tuples of those, etc.)

Example edgelist file format.

With numeric edge data:

```
# read with
# >>> G=nx.read_weighted_edgelist(fh)
# source target data
a b 1
a c 3.14159
d e 42
```

9.2.4 networkx.write_weighted_edgelist

write_weighted_edgelist (*G*, *path*, *comments*='#', *delimiter*=' ')
Write graph G as a list of edges with numeric weights.

Parameters **G** : graph

A NetworkX graph

path : file or string

File or filename to write. Filenames ending in .gz or .bz2 will be compressed.

comments : string, optional

The character used to indicate the start of a comment

delimiter : string, optional

The string used to separate values. The default is whitespace.

See Also:

`read_edgelist`, `write_edgelist`, `write_weighted_edgelist`

Examples

```
>>> import sys
>>> G=nx.Graph()
>>> G.add_edge(1,2,weight=7)
>>> nx.write_weighted_edgelist(G,sys.stdout)
1 2 7
```

9.3 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/>

Requires `pyarsing`: <http://pyarsing.wikispaces.com/>

<code>read_gml(path)</code>	Read graph in GML format from path.
<code>write_gml(G, path)</code>	Write the graph G in GML format to the file or file handle path.
<code>parse_gml(lines)</code>	Parse GML graph from a string or iterable.

9.3.1 `networkx.read_gml`

read_gml (*path*)

Read graph in GML format from path.

Parameters *path* : filename or filehandle

The filename or filehandle to read from.

Returns *G* : MultiGraph or MultiDiGraph

Raises `ImportError` :

If the `pyarsing` module is not available.

See Also:

`write_gml`, `parse_gml`

Notes

This doesn't implement the complete GML specification for nested attributes for graphs, edges, and nodes.

Requires `pyarsing`: <http://pyarsing.wikispaces.com/>

References

GML specification: <http://www.infosun.fim.uni-passau.de/Graphlet/GML/gml-tr.html>

Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_gml(G,'test.gml')
>>> H=nx.read_gml('test.gml')
```

9.3.2 networkx.write_gml

write_gml (*G*, *path*)

Write the graph *G* in GML format to the file or file handle *path*.

Parameters *path* : filename or filehandle

The filename or filehandle to write. Filenames ending in `.gz` or `.gz2` will be compressed.

See Also:

`read_gml`, `parse_gml`

Notes

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.

```
>>> G=nx.path_graph(4)
>>> 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).

For nested attributes for graphs, nodes, and edges you should use dicts for the value of the attribute.

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")
```

9.3.3 networkx.parse_gml

parse_gml (*lines*)

Parse GML graph from a string or iterable.

Parameters **lines** : string or iterable

Data in GML format.

Returns **G** : MultiGraph or MultiDiGraph

Raises **ImportError** :

If the pyparsing module is not available.

See Also:

`write_gml`, `read_gml`

Notes

This stores nested GML attributes as dicts in the NetworkX Graph attribute structures.

Requires pyparsing: <http://pyparsing.wikispaces.com/>

References

GML specification: <http://www.infosun.fim.uni-passau.de/Graphlet/GML/gml-tr.html>

Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_gml(G,'test.gml')
>>> fh=open('test.gml')
>>> H=nx.read_gml(fh)
```

9.4 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 `read_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)`

<code>read_gpickle(path)</code>	Read graph object in Python pickle format
<code>write_gpickle(G, path)</code>	Write graph object in Python pickle format.

9.4.1 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.

9.4.2 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.

9.5 GraphML

Read and write graphs in GraphML format. <http://graphml.graphdrawing.org/>

The module currently supports multi graphs with data but not nested graphs or hypergraphs.

<code>read_graphml(path)</code>	Read graph in GraphML format from path.
<code>parse_graphml(lines)</code>	Read graph in GraphML format from string.

9.5.1 networkx.read_graphml

read_graphml (*path*)

Read graph in GraphML format from path.

Returns a MultiGraph or MultiDiGraph.

Nested graphs and multiple graphs are ignored.

Node, edge and graph data are stored as attributes. The key element can specify the type for each attribute (string, double, etc) and defaults. Data using a key value that matches a key element is given the attribute name `attr.name`. Data whose key value doesn't match a key element uses key value itself as the attribute name.

9.5.2 networkx.parse_graphml

parse_graphml (*lines*)

Read graph in GraphML format from string.

Returns a MultiGraph or MultiDiGraph.

Nested graphs and multiple graphs are ignored.

Node, edge and graph data are stored as attributes. The key element can specify the type for each attribute (string, double, etc) and defaults. Data using a key value that matches a key element is given the attribute name `attr.name`. Data whose key value doesn't match a key element uses key value itself as the attribute name.

9.6 LEDA

Read graphs in LEDA format. See http://www.algorithmic-solutions.info/leda_guide/graphs/leda_native_graph_fileformat.html

<code>read_leda(path)</code>	Read graph in GraphML format from path.
<code>parse_leda(lines)</code>	Parse LEDA.GRAPH format from string or iterable.

9.6.1 networkx.read_leda

read_leda (*path*)

Read graph in GraphML format from path. Returns an XGraph or XDiGraph.

9.6.2 networkx.parse_leda

parse_leda (*lines*)

Parse LEDA.GRAPH format from string or iterable. Returns an Graph or DiGraph.

9.7 YAML

Read and write NetworkX graphs in YAML format. See <http://www.yaml.org> for documentation.

<code>read_yaml(path)</code>	Read graph from YAML format from path.
<code>write_yaml(G, path, **kws[, default_flow_style])</code>	Write graph G in YAML text format to path.

9.7.1 networkx.read_yaml

read_yaml (*path*)

Read graph from YAML format from path.

See <http://www.yaml.org>

9.7.2 networkx.write_yaml

write_yaml (*G, path, default_flow_style=False, **kws*)

Write graph G in YAML text format to path.

See <http://www.yaml.org>

9.8 SparseGraph6

Read graphs in graph6 and sparse6 format. See <http://cs.anu.edu.au/~bdm/data/formats.txt>

<code>read_graph6(path)</code>	Read simple undirected graphs in graph6 format from path.
<code>parse_graph6(str)</code>	Read undirected graph in graph6 format.
<code>read_graph6_list(path)</code>	Read simple undirected graphs in graph6 format from path.
<code>read_sparse6(path)</code>	Read simple undirected graphs in sparse6 format from path.
<code>parse_sparse6(str)</code>	Read undirected graph in sparse6 format.
<code>read_sparse6_list(path)</code>	Read simple undirected graphs in sparse6 format from path.

9.8.1 networkx.read_graph6

read_graph6 (*path*)

Read simple undirected graphs in graph6 format from path. Returns a single Graph.

9.8.2 networkx.parse_graph6

parse_graph6 (*str*)

Read undirected graph in graph6 format.

9.8.3 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.

9.8.4 networkx.read_sparse6

read_sparse6 (*path*)

Read simple undirected graphs in sparse6 format from path. Returns a single Graph.

9.8.5 networkx.parse_sparse6

parse_sparse6 (*str*)

Read undirected graph in sparse6 format.

9.8.6 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.

9.9 Pajek

Read graphs in Pajek format.

See <http://vlado.fmf.uni-lj.si/pub/networks/pajek/doc/draweps.htm> for format information.

This implementation handles only directed and undirected graphs including those with self loops and parallel edges.

<code>read_pajek(path)</code>	Read graph in Pajek format from path.
<code>write_pajek(G, path)</code>	Write in Pajek format to path.
<code>parse_pajek(lines[, edge_attr])</code>	Parse pajek format graph from string or iterable.

9.9.1 networkx.read_pajek

read_pajek (*path*)

Read graph in Pajek format from path.

Returns a MultiGraph or MultiDiGraph.

Parameters `path` : file or string

File or filename to write. Filenames ending in `.gz` or `.bz2` will be compressed.

Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_pajek(G, "test.net")
>>> G=nx.read_pajek("test.net")
```

To create a Graph instead of a MultiGraph use

```
>>> G1=nx.Graph(G)
```

9.9.2 networkx.write_pajek

write_pajek (*G*, *path*)

Write in Pajek format to path.

Parameters `G` : graph

A networkx graph

path : file or string

File or filename to write. Filenames ending in `.gz` or `.bz2` will be compressed.

Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_pajek(G, "test.net")
```

9.9.3 networkx.parse_pajek

parse_pajek (*lines*, *edge_attr=True*)

Parse pajek format graph from string or iterable.

Primarily used as a helper for `read_pajek()`.

See Also:

`read_pajek`

DRAWING

10.1 Matplotlib

Draw networks with matplotlib (pylab).

10.1.1 See Also

matplotlib: <http://matplotlib.sourceforge.net/>

pygraphviz: <http://networkx.lanl.gov/pygraphviz/>

<code>draw(G, **kwds[, pos, ax, hold])</code>	Draw the graph G with Matplotlib (pylab).
<code>draw_networkx(G, **kwds[, pos, with_labels])</code>	Draw the graph G using Matplotlib.
<code>draw_networkx_nodes(G, pos, **kwds[, ...])</code>	Draw the nodes of the graph G.
<code>draw_networkx_edges(G, pos, **kwds[, ...])</code>	Draw the edges of the graph G
<code>draw_networkx_labels(G, pos, **kwds[, ...])</code>	Draw node labels on the graph G
<code>draw_networkx_edge_labels(G, pos, **kwds[, ...])</code>	Draw edge labels.
<code>draw_circular(G, **kwargs)</code>	Draw the graph G with a circular layout
<code>draw_random(G, **kwargs)</code>	Draw the graph G with a random layout.
<code>draw_spectral(G, **kwargs)</code>	Draw the graph G with a spectral layout.
<code>draw_spring(G, **kwargs)</code>	Draw the graph G with a spring layout
<code>draw_shell(G, **kwargs)</code>	Draw networkx graph with shell layout
<code>draw_graphviz(G, **kwargs[, prog])</code>	Draw networkx graph with graphviz layout

10.1.2 networkx.draw

draw (*G*, *pos=None*, *ax=None*, *hold=None*, ***kwds*)

Draw the graph G with Matplotlib (pylab).

Draw the graph as a simple representation with no node labels or edge labels and using the full Matplotlib figure area and no axis labels by default. See `draw_networkx()` for more full-featured drawing that allows title, axis labels etc.

Parameters **G** : graph

A networkx graph

pos : dictionary, optional

A dictionary with nodes as keys and positions as values. If not specified a spring layout positioning will be computed. See `networkx.layout` for functions that compute node positions.

ax : Matplotlib Axes object, optional

Draw the graph in specified Matplotlib axes.

hold: bool, optional :

Set the Matplotlib hold state. If True subsequent draw commands will be added to the current axes.

****kwds**: optional keywords :

See `networkx.draw_networkx()` for a description of optional keywords.

See Also:

`draw_networkx`, `draw_networkx_nodes`, `draw_networkx_edges`, `draw_networkx_labels`,
`draw_networkx_edge_labels`

Notes

This function has the same name as `pylab.draw` and `pyplot.draw` so beware when using

```
>>> from networkx import *
```

since you might overwrite the `pylab.draw` function.

Good alternatives are:

With `pylab`:

```
>>> import pylab as P #
>>> import networkx as nx
>>> G=nx.dodecahedral_graph()
>>> nx.draw(G) # networkx draw()
>>> P.draw() # pylab draw()
```

With `pyplot`

```
>>> import matplotlib.pyplot as plt
>>> import networkx as nx
>>> G=nx.dodecahedral_graph()
>>> nx.draw(G) # networkx draw()
>>> plt.draw() # pyplot draw()
```

Also see the NetworkX drawing examples at <http://networkx.lanl.gov/gallery.html>

Examples

```
>>> G=nx.dodecahedral_graph()
>>> nx.draw(G)
>>> nx.draw(G,pos=nx.spring_layout(G)) # use spring layout
```

10.1.3 networkx.draw_networkx

draw_networkx (*G*, *pos=None*, *with_labels=True*, ***kws*)

Draw the graph *G* using Matplotlib.

Draw the graph with Matplotlib with options for node positions, labeling, titles, and many other drawing features. See `draw()` for simple drawing without labels or axes.

Parameters *G* : graph

A networkx graph

pos : dictionary, optional

A dictionary with nodes as keys and positions as values. If not specified a spring layout positioning will be computed. See `networkx.layout` for functions that compute node positions.

ax : Matplotlib Axes object, optional

Draw the graph in the specified Matplotlib axes.

with_labels: bool, optional :

Set to True (default) to draw labels on the nodes.

nodelist: list, optional :

Draw only specified nodes (default `G.nodes()`)

edgelist: list :

Draw only specified edges (default=`G.edges()`)

node_size: scalar or array :

Size of nodes (default=300). If an array is specified it must be the same length as `nodelist`.

node_color: color string, or array of floats :

Node color. Can be a single color format string (default='r'), or a sequence of colors with the same length as `nodelist`. If numeric values are specified they will be mapped to colors using the `cmap` and `vmin,vmax` parameters. See `matplotlib.scatter` for more details.

node_shape: string :

The shape of the node. Specification is as `matplotlib.scatter` marker, one of 'so^>v<dph8' (default='o').

alpha: float :

The node transparency (default=1.0)

cmap: Matplotlib colormap :

Colormap for mapping intensities of nodes (default=None)

vmin,vmax: floats :

Minimum and maximum for node colormap scaling (default=None)

width: float :

Line width of edges (default =1.0)

edge_color: color string, or array of floats :

Edge color. Can be a single color format string (default='r'), or a sequence of colors with the same length as edgelist. If numeric values are specified they will be mapped to colors using the edge_cmap and edge_vmin,edge_vmax parameters.

edge_cmap: Matplotlib colormap :

Colormap for mapping intensities of edges (default=None)

edge_vmin,edge_vmax: floats :

Minimum and maximum for edge colormap scaling (default=None)

style: string :

Edge line style (default='solid') (solid|dashed|dotted,dashdot)

labels: dictionary :

Node labels in a dictionary keyed by node of text labels (default=None)

font_size: int :

Font size for text labels (default=12)

font_color: string :

Font color string (default='k' black)

font_weight: string :

Font weight (default='normal')

font_family: string :

Font family (default='sans-serif')

See Also:

`draw`, `draw_networkx_nodes`, `draw_networkx_edges`, `draw_networkx_labels`,
`draw_networkx_edge_labels`

Notes

Any keywords not listed above are passed through to `draw_networkx_nodes()`, `draw_networkx_edges()`, and `draw_networkx_labels()`. For finer control of drawing you can call those functions directly.

Examples

```
>>> G=nx.dodecahedral_graph()
>>> nx.draw(G)
>>> nx.draw(G,pos=nx.spring_layout(G)) # use spring layout

>>> import pylab
>>> limits=pylab.axis('off') # turn of axis
```

Also see the NetworkX drawing examples at <http://networkx.lanl.gov/gallery.html>

10.1.4 networkx.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 the nodes of the graph *G*.

This draws only the nodes of the graph *G*.

Parameters **G** : graph

A networkx graph

pos : dictionary

A dictionary with nodes as keys and positions as values. If not specified a spring layout positioning will be computed. See `networkx.layout` for functions that compute node positions.

ax : Matplotlib Axes object, optional

Draw the graph in the specified Matplotlib axes.

nodelist: list, optional :

Draw only specified nodes (default `G.nodes()`)

edgelist: list :

Draw only specified edges (default=`G.edges()`)

node_size: scalar or array :

Size of nodes (default=300). If an array is specified it must be the same length as `nodelist`.

node_color: color string, or array of floats :

Node color. Can be a single color format string (default='r'), or a sequence of colors with the same length as `nodelist`. If numeric values are specified they will be mapped to colors using the `cmap` and `vmin,vmax` parameters. See `matplotlib.scatter` for more details.

node_shape: string :

The shape of the node. Specification is as `matplotlib.scatter` marker, one of 'so^>v<dph8' (default='o').

alpha: float :

The node transparency (default=1.0)

cmap: Matplotlib colormap :

Colormap for mapping intensities of nodes (default=None)

vmin,vmax: floats :

Minimum and maximum for node colormap scaling (default=None)

width': float :

Line width of edges (default =1.0)

See Also:

`draw`, `draw_networkx`, `draw_networkx_edges`, `draw_networkx_labels`,
`draw_networkx_edge_labels`

Notes

Any keywords not listed above are passed through to Matplotlib's scatter function.

Examples

```
>>> G=nx.dodecahedral_graph()
>>> nodes=nx.draw_networkx_nodes(G, pos=nx.spring_layout(G))
```

Also see the NetworkX drawing examples at <http://networkx.lanl.gov/gallery.html>

10.1.5 networkx.draw_networkx_edges

```
draw_networkx_edges(G, pos, edgelist=None, width=1.0, edge_color='k', style='solid', alpha=None,
                    edge_cmap=None, edge_vmin=None, edge_vmax=None, ax=None, arrows=True,
                    **kwargs)
```

Draw the edges of the graph G

This draws only the edges of the graph G.

Parameters **G** : graph

A networkx graph

pos : dictionary

A dictionary with nodes as keys and positions as values. If not specified a spring layout positioning will be computed. See `networkx.layout` for functions that compute node positions.

ax : Matplotlib Axes object, optional

Draw the graph in the specified Matplotlib axes.

alpha: float :

The edge transparency (default=1.0)

width': float :

Line width of edges (default =1.0)

edge_color: color string, or array of floats :

Edge color. Can be a single color format string (default='r'), or a sequence of colors with the same length as `edgelist`. If numeric values are specified they will be mapped to colors using the `edge_cmap` and `edge_vmin,edge_vmax` parameters.

edge_cmap: Matplotlib colormap :

Colormap for mapping intensities of edges (default=None)

edge_vmin,edge_vmax: floats :

Minimum and maximum for edge colormap scaling (default=None)

style: string :

Edge line style (default='solid') (solid|dashed|dotted,dashdot)

See Also:

`draw`, `draw_networkx`, `draw_networkx_nodes`, `draw_networkx_labels`,
`draw_networkx_edge_labels`

Notes

For directed graphs, “arrows” (actually just thicker stubs) are drawn at the head end. Arrows can be turned off with keyword `arrows=False`. Yes, it is ugly but drawing proper arrows with Matplotlib this way is tricky.

Examples

```
>>> G=nx.dodecahedral_graph()
>>> edges=nx.draw_networkx_edges(G,pos=nx.spring_layout(G))
```

Also see the NetworkX drawing examples at <http://networkx.lanl.gov/gallery.html>

10.1.6 networkx.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*, ***kws*)

Draw node labels on the graph *G*

Parameters **G** : graph

A networkx graph

pos : dictionary, optional

A dictionary with nodes as keys and positions as values. If not specified a spring layout positioning will be computed. See `networkx.layout` for functions that compute node positions.

ax : Matplotlib Axes object, optional

Draw the graph in the specified Matplotlib axes.

alpha: float :

The text transparency (default=1.0)

labels: dictionary :

Node labels in a dictionary keyed by node of text labels (default=None)

font_size: int :

Font size for text labels (default=12)

font_color: string :

Font color string (default='k' black)

font_weight: string :

Font weight (default='normal')

font_family: string :

Font family (default='sans-serif')

See Also:

`draw`, `draw_networkx`, `draw_networkx_nodes`, `draw_networkx_edges`,
`draw_networkx_edge_labels`

Examples

```
>>> G=nx.dodecahedral_graph()
>>> labels=nx.draw_networkx_labels(G,pos=nx.spring_layout(G))
```

Also see the NetworkX drawing examples at <http://networkx.lanl.gov/gallery.html>

10.1.7 networkx.draw_networkx_edge_labels

draw_networkx_edge_labels (*G*, *pos*, *edge_labels=None*, *font_size=10*, *font_color='k'*, *font_family='sans-serif'*, *font_weight='normal'*, *alpha=1.0*, *bbox=None*, *ax=None*, ***kwds*)

Draw edge labels.

Parameters **G** : graph

A networkx graph

pos : dictionary, optional

A dictionary with nodes as keys and positions as values. If not specified a spring layout positioning will be computed. See `networkx.layout` for functions that compute node positions.

ax : Matplotlib Axes object, optional

Draw the graph in the specified Matplotlib axes.

alpha: float :

The text transparency (default=1.0)

labels: dictionary :

Node labels in a dictionary keyed by edge two-tuple of text labels (default=None), Only labels for the keys in the dictionary are drawn.

font_size: int :

Font size for text labels (default=12)

font_color: string :

Font color string (default='k' black)

font_weight: string :

Font weight (default='normal')

font_family: string :

Font family (default='sans-serif')

bbox: Matplotlib bbox :

Specify text box shape and colors.

clip_on: bool :

Turn on clipping at axis boundaries (default=True)

See Also:

`draw`, `draw_networkx`, `draw_networkx_nodes`, `draw_networkx_edges`,
`draw_networkx_labels`

Examples

```
>>> G=nx.dodecahedral_graph()
>>> edge_labels=nx.draw_networkx_edge_labels(G,pos=nx.spring_layout(G))
```

Also see the NetworkX drawing examples at <http://networkx.lanl.gov/gallery.html>

10.1.8 networkx.draw_circular

draw_circular (*G*, ***kwargs*)
Draw the graph *G* with a circular layout

10.1.9 networkx.draw_random

draw_random (*G*, ***kwargs*)
Draw the graph *G* with a random layout.

10.1.10 networkx.draw_spectral

draw_spectral (*G*, ***kwargs*)
Draw the graph *G* with a spectral layout.

10.1.11 networkx.draw_spring

draw_spring (*G*, ***kwargs*)
Draw the graph *G* with a spring layout

10.1.12 networkx.draw_shell

draw_shell (*G*, ***kwargs*)
Draw networkx graph with shell layout

10.1.13 networkx.draw_graphviz

draw_graphviz (*G*, *prog='neato'*, ***kwargs*)
Draw networkx graph with graphviz layout

10.2 Graphviz AGraph (dot)

Interface to pygraphviz AGraph class.

10.2.1 Examples

```
>>> G=nx.complete_graph(5)
>>> A=nx.to_agraph(G)
>>> H=nx.from_agraph(A)
```

10.2.2 See Also

Pygraphviz: <http://networkx.lanl.gov/pygraphviz>

<code>from_agraph(A[, create_using])</code>	Return a NetworkX Graph or DiGraph from a PyGraphviz graph.
<code>to_agraph(N)</code>	Return a pygraphviz graph from a NetworkX graph N.
<code>write_dot(G, path)</code>	Write NetworkX graph G to Graphviz dot format on path.
<code>read_dot(path)</code>	Return a NetworkX graph from a dot file on path.
<code>graphviz_layout(G[, prog, root, args])</code>	Create node positions for G using Graphviz.
<code>pygraphviz_layout(G[, prog, root, args])</code>	Create node positions for G using Graphviz.

10.2.3 networkx.from_agraph

from_agraph (*A*, *create_using=None*)

Return a NetworkX Graph or DiGraph from a PyGraphviz graph.

Parameters *A* : PyGraphviz AGraph

A graph created with PyGraphviz

create_using : NetworkX graph class instance

The output is created using the given graph class instance

Notes

The Graph *G* will have a dictionary *G.graph_attr* containing the default graphviz attributes for graphs, nodes and edges.

Default node attributes will be in the dictionary *G.node_attr* which is keyed by node.

Edge attributes will be returned as edge data in *G*. With *edge_attr=False* the edge data will be the Graphviz edge weight attribute or the value 1 if no edge weight attribute is found.

Examples

```
>>> K5=nx.complete_graph(5)
>>> A=nx.to_agraph(K5)
>>> G=nx.from_agraph(A)
>>> G=nx.from_agraph(A)
```

10.2.4 networkx.to_agraph

to_agraph (*N*)

Return a pygraphviz graph from a NetworkX graph *N*.

Parameters **N** : NetworkX graph

A graph created with NetworkX

Notes

If N has an dict N.graph_attr an attempt will be made first to copy properties attached to the graph (see from_agraph) and then updated with the calling arguments if any.

Examples

```
>>> K5=nx.complete_graph(5)
>>> A=nx.to_agraph(K5)
```

10.2.5 networkx.write_dot

write_dot (*G*, *path*)

Write NetworkX graph G to Graphviz dot format on path.

Parameters **G** : graph

A networkx graph

path : filename

Filename or file handle to write.

10.2.6 networkx.read_dot

read_dot (*path*)

Return a NetworkX graph from a dot file on path.

Parameters **path** : file or string

File name or file handle to read.

10.2.7 networkx.graphviz_layout

graphviz_layout (*G*, *prog*='neato', *root*=None, *args*="")

Create node positions for G using Graphviz.

Parameters **G** : NetworkX graph

A graph created with NetworkX

prog : string

Name of Graphviz layout program

root : string, optional

Root node for twopi layout

args : string, optional

Extra arguments to Graphviz layout program

Returns : dictionary

Dictionary of x,y, positions keyed by node.

Notes

This is a wrapper for `pygraphviz_layout`.

Examples

```
>>> G=nx.petersen_graph()
>>> pos=nx.graphviz_layout(G)
>>> pos=nx.graphviz_layout(G,prog='dot')
```

10.2.8 networkx.pygraphviz_layout

pygraphviz_layout (*G*, *prog*='neato', *root*=None, *args*="")
Create node positions for *G* using Graphviz.

Parameters **G** : NetworkX graph

A graph created with NetworkX

prog : string

Name of Graphviz layout program

root : string, optional

Root node for twopi layout

args : string, optional

Extra arguments to Graphviz layout program

Returns : dictionary

Dictionary of x,y, positions keyed by node.

Examples

```
>>> G=nx.petersen_graph()
>>> pos=nx.graphviz_layout(G)
>>> pos=nx.graphviz_layout(G,prog='dot')
```

10.3 Graphviz with pydot

Import and export NetworkX graphs in Graphviz dot format using pydot.

Either this module or `nx_pygraphviz` can be used to interface with graphviz.

10.3.1 See Also

Pydot: <http://www.dkbza.org/pydot.html> Graphviz: <http://www.research.att.com/sw/tools/graphviz/> DOT Language: <http://www.graphviz.org/doc/info/lang.html>

<code>from_pydot(P)</code>	Return a NetworkX graph from a Pydot graph.
<code>to_pydot(N[, strict])</code>	Return a pydot graph from a NetworkX graph N.
<code>write_dot(G, path)</code>	Write NetworkX graph G to Graphviz dot format on path.
<code>read_dot(path)</code>	Return a NetworkX graph from a dot file on path.
<code>graphviz_layout(G[, prog, root, args])</code>	Create node positions for G using Graphviz.
<code>pydot_layout(G, **kwargs[, prog, root])</code>	Create node positions using Pydot and Graphviz.

10.3.2 networkx.from_pydot

from_pydot (*P*)

Return a NetworkX graph from a Pydot graph.

Parameters **P** : Pydot graph

A graph created with Pydot

Examples

```
>>> K5=nx.complete_graph(5)
>>> A=nx.to_pydot(K5)
>>> G=nx.from_pydot(A)
```

10.3.3 networkx.to_pydot

to_pydot (*N, strict=True*)

Return a pydot graph from a NetworkX graph N.

Parameters **N** : NetworkX graph

A graph created with NetworkX

Examples

```
>>> K5=nx.complete_graph(5)
>>> P=nx.to_pydot(K5)
```

10.3.4 networkx.write_dot

write_dot (*G, path*)

Write NetworkX graph G to Graphviz dot format on path.

Parameters **G** : graph

A networkx graph

path : filename

Filename or file handle to write.

10.3.5 networkx.read_dot

read_dot (*path*)

Return a NetworkX graph from a dot file on path.

Parameters **path** : file or string

File name or file handle to read.

10.3.6 networkx.graphviz_layout

graphviz_layout (*G, prog='neato', root=None, args=""*)

Create node positions for G using Graphviz.

Parameters **G** : NetworkX graph

A graph created with NetworkX

prog : string

Name of Graphviz layout program

root : string, optional

Root node for twopi layout

args : string, optional

Extra arguments to Graphviz layout program

Returns : dictionary

Dictionary of x,y, positions keyed by node.

Notes

This is a wrapper for pygraphviz_layout.

Examples

```
>>> G=nx.petersen_graph()
>>> pos=nx.graphviz_layout(G)
>>> pos=nx.graphviz_layout(G,prog='dot')
```

10.3.7 networkx.pydot_layout

pydot_layout (*G, prog='neato', root=None, **kws*)

Create node positions using Pydot and Graphviz.

Returns a dictionary of positions keyed by node.

Examples

```
>>> G=nx.complete_graph(4)
>>> pos=nx.pydot_layout(G)
>>> pos=nx.pydot_layout(G,prog='dot')
```

10.4 Graph Layout

Node positioning algorithms for graph drawing.

<code>circular_layout(G[, dim, scale])</code>	Position nodes on a circle.
<code>random_layout(G[, dim])</code>	
<code>shell_layout(G[, nlist, dim, scale])</code>	Position nodes in concentric circles.
<code>spring_layout(G[, dim, pos, fixed, ...])</code>	Position nodes using Fruchterman-Reingold force-directed algorithm.
<code>spectral_layout(G[, dim, weighted, scale])</code>	Position nodes using the eigenvectors of the graph Laplacian.

10.4.1 networkx.circular_layout

`circular_layout(G, dim=2, scale=1)`

Position nodes on a circle.

Parameters **G** : NetworkX graph

dim : int

Dimension of layout, currently only dim=2 is supported

scale : float

Scale factor for positions

Returns **dict** :

A dictionary of positions keyed by node

Notes

This algorithm currently only works in two dimensions and does not try to minimize edge crossings.

Examples

```
>>> G=nx.path_graph(4)
>>> pos=nx.circular_layout(G)
```

10.4.2 networkx.random_layout

`random_layout(G, dim=2)`

10.4.3 networkx.shell_layout

shell_layout (*G*, *nlist=None*, *dim=2*, *scale=1*)

Position nodes in concentric circles.

Parameters **G** : NetworkX graph

nlist : list of lists

List of node lists for each shell.

dim : int

Dimension of layout, currently only dim=2 is supported

scale : float

Scale factor for positions

Returns **dict** :

A dictionary of positions keyed by node

Notes

This algorithm currently only works in two dimensions and does not try to minimize edge crossings.

Examples

```
>>> G=nx.path_graph(4)
>>> shells=[[0], [1,2,3]]
>>> pos=nx.shell_layout(G, shells)
```

10.4.4 networkx.spring_layout

spring_layout (*G*, *dim=2*, *pos=None*, *fixed=None*, *iterations=50*, *weighted=True*, *scale=1*)

Position nodes using Fruchterman-Reingold force-directed algorithm.

Parameters **G** : NetworkX graph

dim : int

Dimension of layout

pos : dict

Initial positions for nodes as a dictionary with node as keys and values as a list or tuple.

fixed : list

Nodes to keep fixed at initial position.

iterations : int

Number of iterations of spring-force relaxation

weighted : boolean

If True, use edge weights in layout

scale : float

Scale factor for positions

Returns dict :

A dictionary of positions keyed by node

Examples

```
>>> G=nx.path_graph(4)
>>> pos=nx.spring_layout(G)
```

```
# The same using longer function name >>> pos=nx.fruchterman_reingold_layout(G)
```

10.4.5 networkx.spectral_layout

spectral_layout (*G*, *dim=2*, *weighted=True*, *scale=1*)

Position nodes using the eigenvectors of the graph Laplacian.

Parameters G : NetworkX graph

dim : int

Dimension of layout

weighted : boolean

If True, use edge weights in layout

scale : float

Scale factor for positions

Returns dict :

A dictionary of positions keyed by node

Notes

Directed graphs will be considered as undirected graphs when positioning the nodes.

For larger graphs (>500 nodes) this will use the SciPy sparse eigenvalue solver (ARPACK).

Examples

```
>>> G=nx.path_graph(4)
>>> pos=nx.spectral_layout(G)
```


EXCEPTIONS

Base exceptions and errors for NetworkX.

class NetworkXException ()

Base class for exceptions in NetworkX.

class NetworkXError ()

Exception for a serious error in NetworkX

UTILITIES

Helpers for NetworkX.

These are not imported into the base networkx namespace but can be accessed, for example, as

```
>>> import networkx
>>> networkx.utils.is_string_like('spam')
True
```

12.1 Helper functions

<code>is_string_like(obj)</code>	Check if obj is string.
<code>flatten(obj[, result])</code>	Return flattened version of (possibly nested) iterable object.
<code>iterable(obj)</code>	Return True if obj is iterable with a well-defined len()
<code>is_list_of_ints(intlist)</code>	Return True if list is a list of ints.
<code>_get_fh(path[, mode])</code>	Return a file handle for given path.

12.1.1 networkx.utils.is_string_like

is_string_like (*obj*)
Check if obj is string.

12.1.2 networkx.utils.flatten

flatten (*obj, result=None*)
Return flattened version of (possibly nested) iterable object.

12.1.3 networkx.utils.iterable

iterable (*obj*)
Return True if obj is iterable with a well-defined len()

12.1.4 networkx.utils.is_list_of_ints

is_list_of_ints (*intlist*)
Return True if list is a list of ints.

12.1.5 networkx.utils._get_fh

`_get_fh` (*path*, *mode='r'*)

Return a file handle for given path.

Path can be a string or a file handle.

Attempt to uncompress/compress files ending in '.gz' and '.bz2'.

12.2 Data structures and Algorithms

`UnionFind.union(*objects)` Find the sets containing the objects and merge them all.

12.2.1 networkx.utils.UnionFind.union

`union` (**objects*)

Find the sets containing the objects and merge them all.

12.3 Random sequence generators

<code>pareto_sequence</code> (<i>n</i> , <i>exponent</i>)	Return sample sequence of length <i>n</i> from a Pareto distribution.
<code>powerlaw_sequence</code> (<i>n</i> , <i>exponent</i>)	Return sample sequence of length <i>n</i> from a power law distribution.
<code>uniform_sequence</code> (<i>n</i>)	Return sample sequence of length <i>n</i> from a uniform distribution.
<code>cumulative_distribution</code> (<i>distribution</i>)	Return normalized cumulative distribution from discrete distribution.
<code>discrete_sequence</code> (<i>n</i> , <i>distribution</i> , ...)	Return sample sequence of length <i>n</i> from a given discrete distribution or discrete cumulative distribution.

12.3.1 networkx.utils.pareto_sequence

`pareto_sequence` (*n*, *exponent=1.0*)

Return sample sequence of length *n* from a Pareto distribution.

12.3.2 networkx.utils.powerlaw_sequence

`powerlaw_sequence` (*n*, *exponent=2.0*)

Return sample sequence of length *n* from a power law distribution.

12.3.3 networkx.utils.uniform_sequence

`uniform_sequence` (*n*)

Return sample sequence of length *n* from a uniform distribution.

12.3.4 networkx.utils.cumulative_distribution

cumulative_distribution (*distribution*)

Return normalized cumulative distribution from discrete distribution.

12.3.5 networkx.utils.discrete_sequence

discrete_sequence (*n, distribution=None, cdistribution=None*)

Return sample sequence of length *n* from a given discrete distribution or discrete cumulative distribution.

One of the following must be specified.

distribution = histogram of values, will be normalized

cdistribution = normalized discrete cumulative distribution

12.4 SciPy random sequence generators

<code>scipy_pareto_sequence(n[, exponent])</code>	Return sample sequence of length <i>n</i> from a Pareto distribution.
<code>scipy_powerlaw_sequence(n[, exponent])</code>	Return sample sequence of length <i>n</i> from a power law distribution.
<code>scipy_poisson_sequence(n[, mu])</code>	Return sample sequence of length <i>n</i> from a Poisson distribution.
<code>scipy_uniform_sequence(n)</code>	Return sample sequence of length <i>n</i> from a uniform distribution.
<code>scipy_discrete_sequence(n[, distribution])</code>	Return sample sequence of length <i>n</i> from a given discrete distribution

12.4.1 networkx.utils.scipy_pareto_sequence

scipy_pareto_sequence (*n, exponent=1.0*)

Return sample sequence of length *n* from a Pareto distribution.

12.4.2 networkx.utils.scipy_powerlaw_sequence

scipy_powerlaw_sequence (*n, exponent=2.0*)

Return sample sequence of length *n* from a power law distribution.

12.4.3 networkx.utils.scipy_poisson_sequence

scipy_poisson_sequence (*n, mu=1.0*)

Return sample sequence of length *n* from a Poisson distribution.

12.4.4 networkx.utils.scipy_uniform_sequence

scipy_uniform_sequence (*n*)

Return sample sequence of length *n* from a uniform distribution.

12.4.5 networkx.utils.scipy_discrete_sequence

scipy_discrete_sequence (*n*, *distribution=False*)

Return sample sequence of length n from a given discrete distribution

distribution=histogram of values, will be normalized

LICENSE

NetworkX is distributed with the BSD license.

Copyright (C) 2004-2010, NetworkX Developers
Aric Hagberg <hagberg@lanl.gov>
Dan Schult <dschult@colgate.edu>
Pieter Swart <swart@lanl.gov>
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- * Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- * Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- * Neither the name of the NetworkX Developers nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

CITING

To cite NetworkX please use the following publication:

Aric A. Hagberg, Daniel A. Schult and Pieter J. Swart, “Exploring network structure, dynamics, and function using NetworkX”, in *Proceedings of the 7th Python in Science Conference (SciPy2008)*, Gael Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11–15, Aug 2008

CREDITS

NetworkX was originally written by Aric Hagberg, Dan Schult, and Pieter Swart with the help of many others.

Thanks to Guido van Rossum for the idea of using Python for implementing a graph data structure <http://www.python.org/doc/essays/graphs.html>

Thanks to David Eppstein for the idea of representing a graph G so that “for n in G ” loops over the nodes in G and $G[n]$ are node n ’s neighbors.

Thanks to all those who have improved NetworkX by contributing code, bug reports (and fixes), documentation, and input on design, features, and the future of NetworkX.

Thanks especially to the following contributors.

- Katy Bold contributed the Karate Club graph
- Hernan Rozenfeld added `dorogovtsev_goltsev_mendes_graph` and did stress testing
- Brendt Wohlberg added examples from the Stanford GraphBase
- Jim Bagrow reported bugs in the search methods
- Holly Johnsen helped fix the path based centrality measures
- Arnar Flatberg fixed the graph laplacian routines
- Chris Myers suggested using `None` as a default datatype, suggested improvements for the IO routines, added grid generator index tuple labeling and associated routines, and reported bugs
- Joel Miller tested and improved the connected components methods fixed bugs and typos in the graph generators, and contributed the random clustered graph generator.
- Keith Briggs sorted out naming issues for random graphs and wrote `dense_gnm_random_graph`
- Ignacio Rozada provided the Krapivsky-Redner graph generator
- Phillipp Pagel helped fix eccentricity etc. for disconnected graphs
- Sverre Sundsdal contributed bidirectional shortest path and Dijkstra routines, s-metric computation and graph generation
- Ross M. Richardson contributed the expected degree graph generator and helped test the `pygraphviz` interface
- Christopher Ellison implemented the VF2 isomorphism algorithm and contributed the code for matching all the graph types.
- Eben Kenah contributed the strongly connected components and DFS functions.
- Sasha Gutfriend contributed edge betweenness algorithms.
- Udi Weinsberg helped develop intersection and difference operators.

- Matteo Dell'Amico wrote the random regular graph generator.
- Andrew Conway contributed ego_graph, eigenvector centrality, line graph and much more.
- Raf Guns wrote the GraphML writer.
- Salim Fadhley and Matteo Dell'Amico contributed the A* algorithm.
- Fabrice Desclaux contributed the Matplotlib edge labeling code.
- Arpad Horvath fixed the barabasi_albert_graph() generator.
- Minh Van Nguyen contributed the connected_watts_strogatz_graph() and documentation for the Graph and MultiGraph classes.
- Willem Ligtenberg contributed the directed scale free graph generator.
- Loïc Séguin-C. contributed the Ford-Fulkerson max flow and min cut algorithms

GLOSSARY

dictionary FIXME

ebunch An iterable container of edge tuples like a list, iterator, or file.

edge Edges are either two-tuples of nodes (u,v) or three tuples of nodes with an edge attribute dictionary (u,v,dict).

edge attribute Edges can have arbitrary Python objects assigned as attributes by using keyword/value pairs when adding an edge assigning to the `G.edge[u][v]` attribute dictionary for the specified edge u-v.

hashable An object is hashable if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` or `__cmp__()` method). Hashable objects which compare equal must have the same hash value.

Hashability makes an object usable as a dictionary key and a set member, because these data structures use the hash value internally.

All of Python's immutable built-in objects are hashable, while no mutable containers (such as lists or dictionaries) are. Objects which are instances of user-defined classes are hashable by default; they all compare unequal, and their hash value is their `id()`.

Definition from <http://docs.python.org/glossary.html>

nbunch An nbunch is any iterable container of nodes that is not itself a node in the graph. It can be an iterable or an iterator, e.g. a list, set, graph, file, etc..

node A node can be any hashable Python object except None.

node attribute Nodes can have arbitrary Python objects assigned as attributes by using keyword/value pairs when adding a node or assigning to the `G.node[n]` attribute dictionary for the specified node n.

BIBLIOGRAPHY

- [R46] Patrick Doreian, Vladimir Batagelj, and Anuska Ferligoj “Generalized Blockmodeling”, Cambridge University Press, 2004.
- [R35] A Faster Algorithm for Betweenness Centrality. Ulrik Brandes, Journal of Mathematical Sociology 25(2):163-177, 2001. <http://www.inf.uni-konstanz.de/algo/publications/b-fabc-01.pdf>
- [R36] A Faster Algorithm for Betweenness Centrality. Ulrik Brandes, Journal of Mathematical Sociology 25(2):163-177, 2001. <http://www.inf.uni-konstanz.de/algo/publications/b-fabc-01.pdf>
- [R47] Ulrik Brandes and Daniel Fleischer, Centrality Measures Based on Current Flow. Proc. 22nd Symp. Theoretical Aspects of Computer Science (STACS '05). LNCS 3404, pp. 533-544. Springer-Verlag, 2005. <http://www.inf.uni-konstanz.de/algo/publications/bf-cmbcf-05.pdf>
- [R48] Stephenson, K. and Zelen, M. Rethinking centrality: Methods and examples. Social Networks. Volume 11, Issue 1, March 1989, pp. 1-37 [http://dx.doi.org/10.1016/0378-8733\(89\)90016-6](http://dx.doi.org/10.1016/0378-8733(89)90016-6)
- [R37] Centrality Measures Based on Current Flow. Ulrik Brandes and Daniel Fleischer, Proc. 22nd Symp. Theoretical Aspects of Computer Science (STACS '05). LNCS 3404, pp. 533-544. Springer-Verlag, 2005. <http://www.inf.uni-konstanz.de/algo/publications/bf-cmbcf-05.pdf>
- [R38] A measure of betweenness centrality based on random walks, M. E. J. Newman, Social Networks 27, 39-54 (2005).
- [R39] Centrality Measures Based on Current Flow. Ulrik Brandes and Daniel Fleischer, Proc. 22nd Symp. Theoretical Aspects of Computer Science (STACS '05). LNCS 3404, pp. 533-544. Springer-Verlag, 2005. <http://www.inf.uni-konstanz.de/algo/publications/bf-cmbcf-05.pdf>
- [R40] A measure of betweenness centrality based on random walks, M. E. J. Newman, Social Networks 27, 39-54 (2005).
- [R41] Depth-first search and linear graph algorithms, R. Tarjan SIAM Journal of Computing 1(2):146-160, (1972).
- [R42] On finding the strongly connected components in a directed graph. E. Nuutila and E. Soisalon-Soinen Information Processing Letters 49(1): 9-14, (1994)..
- [R43] Depth-first search and linear graph algorithms, R. Tarjan SIAM Journal of Computing 1(2):146-160, (1972).
- [R44] On finding the strongly connected components in a directed graph. E. Nuutila and E. Soisalon-Soinen Information Processing Letters 49(1): 9-14, (1994)..
- [R53] An $O(m)$ Algorithm for Cores Decomposition of Networks Vladimir Batagelj and Matjaz Zaversnik, 2003 <http://arxiv.org/abs/cs.DS/0310049>
- [R95] Skiena, S. S. The Algorithm Design Manual (Springer-Verlag, 1998). http://www.amazon.com/exec/obidos/ASIN/0387948600/ref=ase_thealgorithmrepo/

- [R51] Fleury, “Deux problemes de geometrie de situation”, *Journal de mathematiques elementaires* (1883), 257-261.
- [R52] http://en.wikipedia.org/wiki/Eulerian_path
- [R89] A. Langville and C. Meyer, “A survey of eigenvector methods of web information retrieval.” <http://citeseer.ist.psu.edu/713792.html>
- [R90] Page, Lawrence; Brin, Sergey; Motwani, Rajeev and Winograd, Terry, The PageRank citation ranking: Bringing order to the Web. 1999 <http://dbpubs.stanford.edu:8090/pub/showDoc.Fulltext?lang=en&doc=1999-66&format=pdf>
- [R91] A. Langville and C. Meyer, “A survey of eigenvector methods of web information retrieval.” <http://citeseer.ist.psu.edu/713792.html>
- [R92] Page, Lawrence; Brin, Sergey; Motwani, Rajeev and Winograd, Terry, The PageRank citation ranking: Bringing order to the Web. 1999 <http://dbpubs.stanford.edu:8090/pub/showDoc.Fulltext?lang=en&doc=1999-66&format=pdf>
- [R93] A. Langville and C. Meyer, “A survey of eigenvector methods of web information retrieval.” <http://citeseer.ist.psu.edu/713792.html>
- [R94] Page, Lawrence; Brin, Sergey; Motwani, Rajeev and Winograd, Terry, The PageRank citation ranking: Bringing order to the Web. 1999 <http://dbpubs.stanford.edu:8090/pub/showDoc.Fulltext?lang=en&doc=1999-66&format=pdf>
- [R80] A. Langville and C. Meyer, “A survey of eigenvector methods of web information retrieval.” <http://citeseer.ist.psu.edu/713792.html>
- [R81] Jon Kleinberg, Authoritative sources in a hyperlinked environment *Journal of the ACM* 46 (5): 604-32, 1999. doi:10.1145/324133.324140. <http://www.cs.cornell.edu/home/kleinber/auth.pdf>.
- [R82] A. Langville and C. Meyer, “A survey of eigenvector methods of web information retrieval.” <http://citeseer.ist.psu.edu/713792.html>
- [R83] Jon Kleinberg, Authoritative sources in a hyperlinked environment *Journal of the ACM* 46 (5): 604-32, 1999. doi:10.1145/324133.324140. <http://www.cs.cornell.edu/home/kleinber/auth.pdf>.
- [R84] A. Langville and C. Meyer, “A survey of eigenvector methods of web information retrieval.” <http://citeseer.ist.psu.edu/713792.html>
- [R85] Jon Kleinberg, Authoritative sources in a hyperlinked environment *Journal of the ACM* 46 (5): 604-632, 1999. doi:10.1145/324133.324140. <http://www.cs.cornell.edu/home/kleinber/auth.pdf>.
- [R87] “Efficient Algorithms for Finding Maximum Matching in Graphs” by Zvi Galil, *ACM Computing Surveys*, 1986.
- [R49] M. E. J. Newman, Mixing patterns in networks, *Physical Review E*, 67 026126, 2003
- [R45] M. E. J. Newman, Mixing patterns in networks, *Physical Review E*, 67 026126, 2003
- [R88] M. E. J. Newman, Mixing patterns in networks *Physical Review E*, 67 026126, 2003
- [R50] M. E. J. Newman, Mixing patterns in networks *Physical Review E*, 67 026126, 2003
- [R72] Batagelj and Brandes, “Efficient generation of large random networks”, *Phys. Rev. E*, 71, 036113, 2005.
- [R73] 1. Erdős and A. Rényi, On Random Graphs, *Publ. Math.* 6, 290 (1959).
- [R74] 1. (a) Gilbert, Random Graphs, *Ann. Math. Stat.*, 30, 1141 (1959).
- [R68] 1. Erdős and A. Rényi, On Random Graphs, *Publ. Math.* 6, 290 (1959).
- [R69] 1. (a) Gilbert, Random Graphs, *Ann. Math. Stat.*, 30, 1141 (1959).
- [R67] Donald E. Knuth, *The Art of Computer Programming, Volume 2 / Seminumerical algorithms* Third Edition, Addison-Wesley, 1997.

- [R70] 1. Erdős and A. Rényi, On Random Graphs, *Publ. Math.* 6, 290 (1959).
- [R71] 1. (a) Gilbert, Random Graphs, *Ann. Math. Stat.*, 30, 1141 (1959).
- [R65] 1. Erdős and A. Rényi, On Random Graphs, *Publ. Math.* 6, 290 (1959).
- [R66] 1. (a) Gilbert, Random Graphs, *Ann. Math. Stat.*, 30, 1141 (1959).
- [R75] M. E. J. Newman and D. J. Watts, Renormalization group analysis of the small-world network model, *Physics Letters A*, 263, 341, 1999. [http://dx.doi.org/10.1016/S0375-9601\(99\)00757-4](http://dx.doi.org/10.1016/S0375-9601(99)00757-4)
- [R79] Duncan J. Watts and Steven H. Strogatz, Collective dynamics of small-world networks, *Nature*, 393, pp. 440–442, 1998.
- [R77] A. Steger and N. Wormald, Generating random regular graphs quickly, *Probability and Computing* 8 (1999), 377-396, 1999. <http://citeseer.ist.psu.edu/steger99generating.html>
- [R78] Jeong Han Kim and Van H. Vu, Generating random regular graphs, *Proceedings of the thirty-fifth ACM symposium on Theory of computing*, San Diego, CA, USA, pp 213–222, 2003. <http://doi.acm.org/10.1145/780542.780576>
- [R64] A. L. Barabási and R. Albert “Emergence of scaling in random networks”, *Science* 286, pp 509-512, 1999.
- [R76] P. Holme and B. J. Kim, “Growing scale-free networks with tunable clustering”, *Phys. Rev. E*, 65, 026107, 2002.
- [R54] M.E.J. Newman, “The structure and function of complex networks”, *SIAM REVIEW* 45-2, pp 167-256, 2003.
- [R56] Newman, M. E. J. and Strogatz, S. H. and Watts, D. J. Random graphs with arbitrary degree distributions and their applications *Phys. Rev. E*, 64, 026118 (2001)
- [R57] Fan Chung and L. Lu, Connected components in random graphs with given expected degree sequences, *Ann. Combinatorics*, 6, pp. 125-145, 2002.
- [R58] G. Chartrand and L. Lesniak, “Graphs and Digraphs”, Chapman and Hall/CRC, 1996.
- [R59] G. Chartrand and L. Lesniak, “Graphs and Digraphs”, Chapman and Hall/CRC, 1996.
- [R55] C. Gkantsidis and M. Mihail and E. Zegura, The Markov chain simulation method for generating connected power law random graphs, 2003. <http://citeseer.ist.psu.edu/gkantsidis03markov.html>
- [R60] P. L. Krapivsky and S. Redner, Organization of Growing Random Networks, *Phys. Rev. E*, 63, 066123, 2001.
- [R62] P. L. Krapivsky and S. Redner, Organization of Growing Random Networks, *Phys. Rev. E*, 63, 066123, 2001.
- [R61] P. L. Krapivsky and S. Redner, Network Growth by Copying, *Phys. Rev. E*, 71, 036118, 2005k.},
- [R63] B. Bollobás, C. Borgs, J. Chayes, and O. Riordan, Directed scale-free graphs, *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, 132–139, 2003.
- [R86] Fan Chung-Graham, *Spectral Graph Theory*, CBMS Regional Conference Series in Mathematics, Number 92, 1997.

MODULE INDEX

N

networkx.algorithms.bipartite, 133
networkx.algorithms.block, 135
networkx.algorithms.boundary, 136
networkx.algorithms.centralities, 137
networkx.algorithms.centralities.betweenness, 139
networkx.algorithms.centralities.closeness, 138
networkx.algorithms.centralities.current_flow_betweenness, 141
networkx.algorithms.centralities.current_flow_closeness, 141
networkx.algorithms.centralities.eigenvector, 143
networkx.algorithms.centralities.load, 144
networkx.algorithms.clique, 145
networkx.algorithms.cluster, 147
networkx.algorithms.components, 150
networkx.algorithms.components.attracting, 157
networkx.algorithms.components.connected, 150
networkx.algorithms.components.strongly_connected, 153
networkx.algorithms.components.weakly_connected, 156
networkx.algorithms.core, 158
networkx.algorithms.cycles, 158
networkx.algorithms.dag, 159
networkx.algorithms.distance_measures, 161
networkx.algorithms.euler, 162
networkx.algorithms.flow, 164
networkx.algorithms.isolates, 166
networkx.algorithms.link_analysis.hits_alg, 182
networkx.algorithms.link_analysis.pagerank_alg, 179
networkx.algorithms.matching, 185
networkx.algorithms.mixing, 186
networkx.algorithms.mst, 191
networkx.algorithms.operators, 191
networkx.algorithms.shortest_paths.astar, 208
networkx.algorithms.shortest_paths.generic, 195
networkx.algorithms.shortest_paths.unweighted, 198
networkx.algorithms.shortest_paths.weighted, 201
networkx.algorithms.traversal.depth_first_search, 209
networkx.classes.function, 213
networkx.convert, 259
networkx.drawing.layout, 299
networkx.drawing.nx_agraph, 293
networkx.drawing.nx_pydot, 296
networkx.drawing.nx_pylab, 285
networkx.exception, 303
networkx.generators.atlas, 217
networkx.generators.bipartite, 247
networkx.generators.classic, 217
networkx.generators.degree_seq, 236
networkx.generators.directed, 243
networkx.generators.ego, 252
networkx.generators.geometric, 247
networkx.generators.hybrid, 247
networkx.generators.line, 251
networkx.generators.random_graphs, 226
networkx.generators.small, 221
networkx.generators.stochastic, 252
networkx.linalg.attrmatrix, 255
networkx.linalg.spectrum, 253
networkx.readwrite.adjlist, 269
networkx.readwrite.edgelist, 273
networkx.readwrite.gml, 277
networkx.readwrite.gpickle, 279
networkx.readwrite.graphml, 280
networkx.readwrite.leda, 281
networkx.readwrite.nx_yaml, 281

`networkx.readwrite.pajek`, 282
`networkx.readwrite.sparsegraph6`, 281
`networkx.utils`, 305

INDEX

Symbols

`__contains__()` (DiGraph method), 58
`__contains__()` (Graph method), 28
`__contains__()` (MultiDiGraph method), 120
`__contains__()` (MultiGraph method), 89
`__getitem__()` (DiGraph method), 54
`__getitem__()` (Graph method), 26
`__getitem__()` (MultiDiGraph method), 117
`__getitem__()` (MultiGraph method), 86
`__init__()` (DiGraph method), 40
`__init__()` (DiGraphMatcher method), 171
`__init__()` (Graph method), 12
`__init__()` (GraphMatcher method), 169
`__init__()` (MultiDiGraph method), 100
`__init__()` (MultiGraph method), 71
`__init__()` (WeightedDiGraphMatcher method), 175
`__init__()` (WeightedGraphMatcher method), 173
`__init__()` (WeightedMultiDiGraphMatcher method), 178
`__init__()` (WeightedMultiGraphMatcher method), 176
`__iter__()` (DiGraph method), 49
`__iter__()` (Graph method), 22
`__iter__()` (MultiDiGraph method), 111
`__iter__()` (MultiGraph method), 82
`__len__()` (DiGraph method), 59
`__len__()` (Graph method), 30
`__len__()` (MultiDiGraph method), 122
`__len__()` (MultiGraph method), 90
`__get_fh()` (in module `networkx.utils`), 306

A

`add_cycle()` (DiGraph method), 47
`add_cycle()` (Graph method), 20
`add_cycle()` (MultiDiGraph method), 108
`add_cycle()` (MultiGraph method), 80
`add_edge()` (DiGraph method), 43
`add_edge()` (Graph method), 15
`add_edge()` (MultiDiGraph method), 103
`add_edge()` (MultiGraph method), 75
`add_edges_from()` (DiGraph method), 44
`add_edges_from()` (Graph method), 16
`add_edges_from()` (MultiDiGraph method), 104

`add_edges_from()` (MultiGraph method), 76
`add_node()` (DiGraph method), 40
`add_node()` (Graph method), 13
`add_node()` (MultiDiGraph method), 101
`add_node()` (MultiGraph method), 72
`add_nodes_from()` (DiGraph method), 41
`add_nodes_from()` (Graph method), 14
`add_nodes_from()` (MultiDiGraph method), 102
`add_nodes_from()` (MultiGraph method), 73
`add_path()` (DiGraph method), 47
`add_path()` (Graph method), 19
`add_path()` (MultiDiGraph method), 108
`add_path()` (MultiGraph method), 79
`add_star()` (DiGraph method), 46
`add_star()` (Graph method), 19
`add_star()` (MultiDiGraph method), 107
`add_star()` (MultiGraph method), 79
`add_weighted_edges_from()` (DiGraph method), 44
`add_weighted_edges_from()` (Graph method), 17
`add_weighted_edges_from()` (MultiDiGraph method),
105
`add_weighted_edges_from()` (MultiGraph method), 77
`adj_matrix()` (in module `networkx.linalg.spectrum`), 253
`adjacency_iter()` (DiGraph method), 56
`adjacency_iter()` (Graph method), 27
`adjacency_iter()` (MultiDiGraph method), 118
`adjacency_iter()` (MultiGraph method), 87
`adjacency_list()` (DiGraph method), 56
`adjacency_list()` (Graph method), 26
`adjacency_list()` (MultiDiGraph method), 118
`adjacency_list()` (MultiGraph method), 87
`adjacency_spectrum()` (in module `networkx.linalg.spectrum`), 255
`all_pairs_dijkstra_path()` (in module `networkx`), 204
`all_pairs_dijkstra_path_length()` (in module `networkx`),
205
`all_pairs_shortest_path()` (in module `networkx`), 199
`all_pairs_shortest_path_length()` (in module `networkx`),
200
`astar_path()` (in module `networkx`), 208
`astar_path_length()` (in module `networkx`), 209
`attr_matrix()` (in module `networkx.linalg.attrmatrix`), 255

- attr_sparse_matrix() (in module networkx.linalg.attrmatrix), 257
- attracting_component_subgraphs() (in module networkx.algorithms.components.attracting), 158
- attracting_components() (in module networkx.algorithms.components.attracting), 157
- attribute_assortativity() (in module networkx), 187
- attribute_mixing_dict() (in module networkx), 190
- attribute_mixing_matrix() (in module networkx), 189
- authority_matrix() (in module networkx), 185
- average_clustering() (in module networkx), 149
- average_shortest_path_length() (in module networkx), 197
- ## B
- balanced_tree() (in module networkx.generators.classic), 218
- barabasi_albert_graph() (in module networkx.generators.random_graphs), 233
- barbell_graph() (in module networkx.generators.classic), 218
- betweenness_centrality() (in module networkx.algorithms.centrality.betweenness), 139
- bidirectional_dijkstra() (in module networkx), 206
- bidirectional_shortest_path() (in module networkx), 207
- binomial_graph() (in module networkx.generators.random_graphs), 230
- bipartite_alternating_havel_hakimi_graph() (in module networkx.generators.bipartite), 249
- bipartite_color() (in module networkx), 134
- bipartite_configuration_model() (in module networkx.generators.bipartite), 248
- bipartite_havel_hakimi_graph() (in module networkx.generators.bipartite), 248
- bipartite_preferential_attachment_graph() (in module networkx.generators.bipartite), 250
- bipartite_random_regular_graph() (in module networkx.generators.bipartite), 250
- bipartite_reverse_havel_hakimi_graph() (in module networkx.generators.bipartite), 249
- bipartite_sets() (in module networkx), 133
- blockmodel() (in module networkx), 135
- bull_graph() (in module networkx.generators.small), 223
- ## C
- candidate_pairs_iter() (DiGraphMatcher method), 172
- candidate_pairs_iter() (GraphMatcher method), 170
- candidate_pairs_iter() (WeightedDiGraphMatcher method), 176
- candidate_pairs_iter() (WeightedGraphMatcher method), 174
- candidate_pairs_iter() (WeightedMultiDiGraphMatcher method), 179
- candidate_pairs_iter() (WeightedMultiGraphMatcher method), 177
- cartesian_product() (in module networkx), 192
- center() (in module networkx), 161
- chvatal_graph() (in module networkx.generators.small), 223
- circular_ladder_graph() (in module networkx.generators.classic), 219
- circular_layout() (in module networkx), 299
- clear() (DiGraph method), 47
- clear() (Graph method), 20
- clear() (MultiDiGraph method), 109
- clear() (MultiGraph method), 80
- cliques_containing_node() (in module networkx), 147
- closeness_centrality() (in module networkx), 139
- closeness_vitality() (in module networkx), 210
- clustering() (in module networkx), 149
- complement() (in module networkx), 193
- complete_bipartite_graph() (in module networkx.generators.classic), 219
- complete_graph() (in module networkx.generators.classic), 219
- compose() (in module networkx), 192
- condensation() (in module networkx.algorithms.components.strongly_connected), 155
- configuration_model() (in module networkx.generators.degree_seq), 237
- connected_component_subgraphs() (in module networkx.algorithms.components.connected), 152
- connected_components() (in module networkx.algorithms.components.connected), 151
- connected_double_edge_swap() (in module networkx.generators.degree_seq), 242
- connected_watts_strogatz_graph() (in module networkx.generators.random_graphs), 232
- convert_node_labels_to_integers() (in module networkx.convert), 260
- copy() (DiGraph method), 66
- copy() (Graph method), 34
- copy() (MultiDiGraph method), 129
- copy() (MultiGraph method), 95
- could_be_isomorphic() (in module networkx), 168
- create_degree_sequence() (in module networkx.generators.degree_seq), 241
- create_empty_copy() (in module networkx), 215
- cubical_graph() (in module networkx.generators.small), 223
- cumulative_distribution() (in module networkx.utils), 307
- current_flow_betweenness_centrality() (in module net-

- workx.algorithms centrality.current_flow_betweenness_centrality() (in module networkx), 142
 current_flow_closeness_centrality() (in module networkx), 141
 cycle_basis() (in module networkx), 159
 cycle_graph() (in module networkx.generators.classic), 219
- ## D
- degree() (DiGraph method), 60
 degree() (Graph method), 30
 degree() (MultiDiGraph method), 122
 degree() (MultiGraph method), 91
 degree_assortativity() (in module networkx), 186
 degree_centrality() (in module networkx), 137
 degree_histogram() (in module networkx), 214
 degree_iter() (DiGraph method), 60
 degree_iter() (Graph method), 31
 degree_iter() (MultiDiGraph method), 123
 degree_iter() (MultiGraph method), 91
 degree_mixing_dict() (in module networkx), 190
 degree_mixing_matrix() (in module networkx), 190
 degree_pearsonr() (in module networkx), 188
 degree_sequence_tree() (in module networkx.generators.degree_seq), 240
 dense_gnm_random_graph() (in module networkx.generators.random_graphs), 228
 density() (in module networkx), 213
 desargues_graph() (in module networkx.generators.small), 223
 dfs_postorder() (in module networkx), 210
 dfs_predecessor() (in module networkx), 210
 dfs_preorder() (in module networkx), 210
 dfs_successor() (in module networkx), 210
 dfs_tree() (in module networkx), 210
 diameter() (in module networkx), 161
 diamond_graph() (in module networkx.generators.small), 223
 dictionary, 315
 difference() (in module networkx), 194
 DiGraph() (in module networkx), 37
 dijkstra_path() (in module networkx), 202
 dijkstra_path_length() (in module networkx), 202
 dijkstra_predecessor_and_distance() (in module networkx), 208
 directed_configuration_model() (in module networkx.generators.degree_seq), 238
 directed_gnp_random_graph() (in module networkx.generators.random_graphs), 227
 discrete_sequence() (in module networkx.utils), 307
 disjoint_union() (in module networkx), 194
 dodecahedral_graph() (in module networkx.generators.small), 224
- double_edge_swap() (in module networkx.generators.degree_seq), 241
 draw() (in module networkx), 285
 draw_circular() (in module networkx), 293
 draw_graphviz() (in module networkx), 293
 draw_networkx() (in module networkx), 287
 draw_networkx_edge_labels() (in module networkx), 292
 draw_networkx_edges() (in module networkx), 290
 draw_networkx_labels() (in module networkx), 291
 draw_networkx_nodes() (in module networkx), 289
 draw_random() (in module networkx), 293
 draw_shell() (in module networkx), 293
 draw_spectral() (in module networkx), 293
 draw_spring() (in module networkx), 293
- ## E
- ebunch, 315
 eccentricity() (in module networkx), 161
 edge, 315
 edge attribute, 315
 edge_betweenness_centrality() (in module networkx.algorithms.centrality.betweenness), 140
 edge_boundary() (in module networkx), 136
 edge_current_flow_betweenness_centrality() (in module networkx.algorithms.centrality.current_flow_betweenness), 142
 edge_load() (in module networkx.algorithms.centrality.load), 145
 edges() (DiGraph method), 50
 edges() (Graph method), 22
 edges() (MultiDiGraph method), 112
 edges() (MultiGraph method), 82
 edges_iter() (DiGraph method), 51
 edges_iter() (Graph method), 23
 edges_iter() (MultiDiGraph method), 113
 edges_iter() (MultiGraph method), 83
 ego_graph() (in module networkx.generators.ego), 252
 eigenvector_centrality() (in module networkx), 143
 eigenvector_centrality_numpy() (in module networkx), 144
 empty_graph() (in module networkx.generators.classic), 219
 erdos_renyi_graph() (in module networkx.generators.random_graphs), 229
 eulerian_circuit() (in module networkx), 163
 expected_degree_graph() (in module networkx.generators.degree_seq), 239
- ## F
- fast_could_be_isomorphic() (in module networkx), 169

- fast_gnp_random_graph() (in module networkx.generators.random_graphs), 226
 faster_could_be_isomorphic() (in module networkx), 169
 find_cliques() (in module networkx), 146
 find_cores() (in module networkx), 158
 flatten() (in module networkx.utils), 305
 floyd_warshall() (in module networkx), 201
 ford_fulkerson() (in module networkx), 166
 freeze() (in module networkx), 214
 from_agraph() (in module networkx), 294
 from_dict_of_dicts() (in module networkx.convert), 262
 from_dict_of_lists() (in module networkx.convert), 263
 from_edgelist() (in module networkx.convert), 264
 from_numpy_matrix() (in module networkx.convert), 265
 from_pydot() (in module networkx), 297
 from_scipy_sparse_matrix() (in module networkx.convert), 266
 frucht_graph() (in module networkx.generators.small), 224
- ## G
- get_edge_data() (DiGraph method), 53
 get_edge_data() (Graph method), 24
 get_edge_data() (MultiDiGraph method), 116
 get_edge_data() (MultiGraph method), 84
 gn_graph() (in module networkx.generators.directed), 244
 gnc_graph() (in module networkx.generators.directed), 245
 gnm_random_graph() (in module networkx.generators.random_graphs), 229
 gnp_random_graph() (in module networkx.generators.random_graphs), 227
 gnr_graph() (in module networkx.generators.directed), 245
 google_matrix() (in module networkx), 182
 Graph() (in module networkx), 9
 graph_atlas_g() (in module networkx.generators.atlas), 217
 graph_clique_number() (in module networkx), 147
 graph_number_of_cliques() (in module networkx), 147
 graphviz_layout() (in module networkx), 295, 298
 grid_2d_graph() (in module networkx.generators.classic), 220
 grid_graph() (in module networkx.generators.classic), 220
- ## H
- has_edge() (DiGraph method), 58
 has_edge() (Graph method), 29
 has_edge() (MultiDiGraph method), 121
 has_edge() (MultiGraph method), 89
 has_node() (DiGraph method), 57
 has_node() (Graph method), 28
 has_node() (MultiDiGraph method), 120
 has_node() (MultiGraph method), 88
 hashable, 315
 havel_hakimi_graph() (in module networkx.generators.degree_seq), 240
 heawood_graph() (in module networkx.generators.small), 224
 hits() (in module networkx), 182
 hits_numpy() (in module networkx), 183
 hits_scipy() (in module networkx), 184
 house_graph() (in module networkx.generators.small), 224
 house_x_graph() (in module networkx.generators.small), 224
 hub_matrix() (in module networkx), 185
 hypercube_graph() (in module networkx.generators.classic), 220
- ## I
- icosahedral_graph() (in module networkx.generators.small), 224
 in_degree() (DiGraph method), 61
 in_degree() (MultiDiGraph method), 124
 in_degree_centrality() (in module networkx), 138
 in_degree_iter() (DiGraph method), 61
 in_degree_iter() (MultiDiGraph method), 124
 in_edges() (DiGraph method), 53
 in_edges() (MultiDiGraph method), 115
 in_edges_iter() (DiGraph method), 53
 in_edges_iter() (MultiDiGraph method), 115
 info() (in module networkx), 213
 initialize() (DiGraphMatcher method), 172
 initialize() (GraphMatcher method), 170
 initialize() (WeightedDiGraphMatcher method), 175
 initialize() (WeightedGraphMatcher method), 174
 initialize() (WeightedMultiDiGraphMatcher method), 178
 initialize() (WeightedMultiGraphMatcher method), 177
 intersection() (in module networkx), 194
 is_attracting_component() (in module networkx.algorithms.components.attracting), 157
 is_bipartite() (in module networkx), 133
 is_connected() (in module networkx.algorithms.components.connected), 150
 is_directed_acyclic_graph() (in module networkx), 160
 is_eulerian() (in module networkx), 163
 is_frozen() (in module networkx), 215
 is_isolate() (in module networkx), 167
 is_isomorphic() (DiGraphMatcher method), 172
 is_isomorphic() (GraphMatcher method), 170
 is_isomorphic() (in module networkx), 168
 is_isomorphic() (WeightedDiGraphMatcher method), 175

- is_isomorphic() (WeightedGraphMatcher method), 174
 is_isomorphic() (WeightedMultiDiGraphMatcher method), 178
 is_isomorphic() (WeightedMultiGraphMatcher method), 177
 is_kl_connected() (in module networkx.generators.hybrid), 247
 is_list_of_ints() (in module networkx.utils), 305
 is_string_like() (in module networkx.utils), 305
 is_strongly_connected() (in module networkx.algorithms.components.strongly_connected), 153
 is_valid_degree_sequence() (in module networkx.generators.degree_seq), 241
 is_weakly_connected() (in module networkx.algorithms.components.weakly_connected), 156
 isolates() (in module networkx), 167
 isomorphisms_iter() (DiGraphMatcher method), 172
 isomorphisms_iter() (GraphMatcher method), 170
 isomorphisms_iter() (WeightedDiGraphMatcher method), 175
 isomorphisms_iter() (WeightedGraphMatcher method), 174
 isomorphisms_iter() (WeightedMultiDiGraphMatcher method), 179
 isomorphisms_iter() (WeightedMultiGraphMatcher method), 177
 iterable() (in module networkx.utils), 305
- ## K
- kl_connected_subgraph() (in module networkx.generators.hybrid), 247
 kosaraju_strongly_connected_components() (in module networkx.algorithms.components.strongly_connected), 155
 krackhardt_kite_graph() (in module networkx.generators.small), 224
- ## L
- ladder_graph() (in module networkx.generators.classic), 220
 laplacian() (in module networkx.linalg.spectrum), 254
 laplacian_spectrum() (in module networkx.linalg.spectrum), 254
 LCF_graph() (in module networkx.generators.small), 222
 li_smax_graph() (in module networkx.generators.degree_seq), 242
 line_graph() (in module networkx.generators.line), 251
 load_centrality() (in module networkx.algorithms.centralty.load), 145
 lollipop_graph() (in module networkx.generators.classic), 220
- ## M
- make_clique_bipartite() (in module networkx), 146
 make_max_clique_graph() (in module networkx), 146
 make_small_graph() (in module networkx.generators.small), 222
 match() (DiGraphMatcher method), 172
 match() (GraphMatcher method), 170
 match() (WeightedDiGraphMatcher method), 176
 match() (WeightedGraphMatcher method), 174
 match() (WeightedMultiDiGraphMatcher method), 179
 match() (WeightedMultiGraphMatcher method), 177
 max_flow() (in module networkx), 164
 max_weight_matching() (in module networkx), 185
 min_cut() (in module networkx), 165
 minimum_spanning_tree() (in module networkx), 191
 moebius_kantor_graph() (in module networkx.generators.small), 224
 MultiDiGraph() (in module networkx), 97
 MultiGraph() (in module networkx), 68
- ## N
- nbunch, 315
 nbunch_iter() (DiGraph method), 56
 nbunch_iter() (Graph method), 27
 nbunch_iter() (MultiDiGraph method), 119
 nbunch_iter() (MultiGraph method), 87
 neighbor_connectivity() (in module networkx), 188
 neighbors() (DiGraph method), 54
 neighbors() (Graph method), 25
 neighbors() (MultiDiGraph method), 116
 neighbors() (MultiGraph method), 85
 neighbors_iter() (DiGraph method), 54
 neighbors_iter() (Graph method), 25
 neighbors_iter() (MultiDiGraph method), 117
 neighbors_iter() (MultiGraph method), 86
 networkx.algorithms.bipartite (module), 133
 networkx.algorithms.block (module), 135
 networkx.algorithms.boundary (module), 136
 networkx.algorithms.centralty (module), 137
 networkx.algorithms.centralty.betweenness (module), 139
 networkx.algorithms.centralty.closeness (module), 138
 networkx.algorithms.centralty.current_flow_betweenness (module), 141
 networkx.algorithms.centralty.current_flow_closeness (module), 141
 networkx.algorithms.centralty.eigenvector (module), 143
 networkx.algorithms.centralty.load (module), 144
 networkx.algorithms.clique (module), 145
 networkx.algorithms.cluster (module), 147
 networkx.algorithms.components (module), 150
 networkx.algorithms.components.attracting (module), 157

- networkx.algorithms.components.connected (module), 150
- networkx.algorithms.components.strongly_connected (module), 153
- networkx.algorithms.components.weakly_connected (module), 156
- networkx.algorithms.core (module), 158
- networkx.algorithms.cycles (module), 158
- networkx.algorithms.dag (module), 159
- networkx.algorithms.distance_measures (module), 161
- networkx.algorithms.euler (module), 162
- networkx.algorithms.flow (module), 164
- networkx.algorithms.isolates (module), 166
- networkx.algorithms.link_analysis.hits_alg (module), 182
- networkx.algorithms.link_analysis.pagerank_alg (module), 179
- networkx.algorithms.matching (module), 185
- networkx.algorithms.mixing (module), 186
- networkx.algorithms.mst (module), 191
- networkx.algorithms.operators (module), 191
- networkx.algorithms.shortest_paths.astar (module), 208
- networkx.algorithms.shortest_paths.generic (module), 195
- networkx.algorithms.shortest_paths.unweighted (module), 198
- networkx.algorithms.shortest_paths.weighted (module), 201
- networkx.algorithms.traversal.depth_first_search (module), 209
- networkx.algorithms.vitality (module), 210
- networkx.classes.function (module), 213
- networkx.convert (module), 259
- networkx.drawing.layout (module), 299
- networkx.drawing.nx_agraph (module), 293
- networkx.drawing.nx_pydot (module), 296
- networkx.drawing.nx_pylab (module), 285
- networkx.exception (module), 303
- networkx.generators.atlas (module), 217
- networkx.generators.bipartite (module), 247
- networkx.generators.classic (module), 217
- networkx.generators.degree_seq (module), 236
- networkx.generators.directed (module), 243
- networkx.generators.ego (module), 252
- networkx.generators.geometric (module), 247
- networkx.generators.hybrid (module), 247
- networkx.generators.line (module), 251
- networkx.generators.random_graphs (module), 226
- networkx.generators.small (module), 221
- networkx.generators.stochastic (module), 252
- networkx.linalg.attrmatrix (module), 255
- networkx.linalg.spectrum (module), 253
- networkx.readwrite.adjlist (module), 269
- networkx.readwrite.edgelist (module), 273
- networkx.readwrite.gml (module), 277
- networkx.readwrite.gpickle (module), 279
- networkx.readwrite.graphml (module), 280
- networkx.readwrite.leda (module), 281
- networkx.readwrite.nx_yaml (module), 281
- networkx.readwrite.pajek (module), 282
- networkx.readwrite.sparsegraph6 (module), 281
- networkx.utils (module), 305
- NetworkXError (class in networkx), 303
- NetworkXException (class in networkx), 303
- newman_watts_strogatz_graph() (in module networkx.generators.random_graphs), 230
- node, 315
- node attribute, 315
- node_boundary() (in module networkx), 137
- node_clique_number() (in module networkx), 147
- node_connected_component() (in module networkx.algorithms.components.connected), 152
- nodes() (DiGraph method), 48
- nodes() (Graph method), 21
- nodes() (MultiDiGraph method), 110
- nodes() (MultiGraph method), 81
- nodes_iter() (DiGraph method), 49
- nodes_iter() (Graph method), 21
- nodes_iter() (MultiDiGraph method), 111
- nodes_iter() (MultiGraph method), 81
- nodes_with_selfloops() (DiGraph method), 64
- nodes_with_selfloops() (Graph method), 33
- nodes_with_selfloops() (MultiDiGraph method), 127
- nodes_with_selfloops() (MultiGraph method), 93
- normalized_laplacian() (in module networkx.linalg.spectrum), 254
- null_graph() (in module networkx.generators.classic), 221
- number_attracting_components() (in module networkx.algorithms.components.attracting), 157
- number_connected_components() (in module networkx.algorithms.components.connected), 151
- number_of_cliques() (in module networkx), 147
- number_of_edges() (DiGraph method), 64
- number_of_edges() (Graph method), 32
- number_of_edges() (MultiDiGraph method), 126
- number_of_edges() (MultiGraph method), 92
- number_of_nodes() (DiGraph method), 59
- number_of_nodes() (Graph method), 30
- number_of_nodes() (MultiDiGraph method), 122
- number_of_nodes() (MultiGraph method), 90
- number_of_selfloops() (DiGraph method), 65
- number_of_selfloops() (Graph method), 34
- number_of_selfloops() (MultiDiGraph method), 128
- number_of_selfloops() (MultiGraph method), 94

- number_strongly_connected_components()
(in module `networkx.algorithms.components.strongly_connected`), 153
- number_weakly_connected_components()
(in module `networkx.algorithms.components.weakly_connected`), 156
- numeric_assortativity() (in module `networkx`), 187
- ## O
- octahedral_graph() (in module `networkx.generators.small`), 225
- order() (DiGraph method), 59
- order() (Graph method), 29
- order() (MultiDiGraph method), 121
- order() (MultiGraph method), 90
- out_degree() (DiGraph method), 62
- out_degree() (MultiDiGraph method), 125
- out_degree_centrality() (in module `networkx`), 138
- out_degree_iter() (DiGraph method), 63
- out_degree_iter() (MultiDiGraph method), 125
- out_edges() (DiGraph method), 51
- out_edges() (MultiDiGraph method), 113
- out_edges_iter() (DiGraph method), 52
- out_edges_iter() (MultiDiGraph method), 114
- ## P
- pagerank() (in module `networkx`), 179
- pagerank_numpy() (in module `networkx`), 180
- pagerank_scipy() (in module `networkx`), 181
- pappus_graph() (in module `networkx.generators.small`), 225
- pareto_sequence() (in module `networkx.utils`), 306
- parse_gml() (in module `networkx`), 279
- parse_graph6() (in module `networkx`), 282
- parse_graphml() (in module `networkx`), 280
- parse_leda() (in module `networkx`), 281
- parse_pajek() (in module `networkx`), 283
- parse_sparse6() (in module `networkx`), 282
- path_graph() (in module `networkx.generators.classic`), 221
- periphery() (in module `networkx`), 162
- petersen_graph() (in module `networkx.generators.small`), 225
- powerlaw_cluster_graph() (in module `networkx.generators.random_graphs`), 234
- powerlaw_sequence() (in module `networkx.utils`), 306
- predecessor() (in module `networkx`), 200
- predecessors() (DiGraph method), 55
- predecessors() (MultiDiGraph method), 118
- predecessors_iter() (DiGraph method), 55
- predecessors_iter() (MultiDiGraph method), 118
- project() (in module `networkx`), 134
- pydot_layout() (in module `networkx`), 298
- pygraphviz_layout() (in module `networkx`), 296
- ## R
- radius() (in module `networkx`), 162
- random_geometric_graph() (in module `networkx.generators.geometric`), 247
- random_layout() (in module `networkx`), 299
- random_lobster() (in module `networkx.generators.random_graphs`), 234
- random_powerlaw_tree() (in module `networkx.generators.random_graphs`), 235
- random_powerlaw_tree_sequence() (in module `networkx.generators.random_graphs`), 236
- random_regular_graph() (in module `networkx.generators.random_graphs`), 232
- random_shell_graph() (in module `networkx.generators.random_graphs`), 235
- read_adjlist() (in module `networkx`), 269
- read_dot() (in module `networkx`), 295, 298
- read_edgelist() (in module `networkx`), 273
- read_gml() (in module `networkx`), 277
- read_gpickle() (in module `networkx`), 280
- read_graph6() (in module `networkx`), 282
- read_graph6_list() (in module `networkx`), 282
- read_graphml() (in module `networkx`), 280
- read_leda() (in module `networkx`), 281
- read_multiline_adjlist() (in module `networkx`), 271
- read_pajek() (in module `networkx`), 282
- read_sparse6() (in module `networkx`), 282
- read_sparse6_list() (in module `networkx`), 282
- read_weighted_edgelist() (in module `networkx`), 276
- read_yaml() (in module `networkx`), 281
- relabel_nodes() (in module `networkx.convert`), 261
- remove_edge() (DiGraph method), 45
- remove_edge() (Graph method), 18
- remove_edge() (MultiDiGraph method), 106
- remove_edge() (MultiGraph method), 77
- remove_edges_from() (DiGraph method), 46
- remove_edges_from() (Graph method), 18
- remove_edges_from() (MultiDiGraph method), 107
- remove_edges_from() (MultiGraph method), 78
- remove_node() (DiGraph method), 42
- remove_node() (Graph method), 14
- remove_node() (MultiDiGraph method), 102
- remove_node() (MultiGraph method), 74
- remove_nodes_from() (DiGraph method), 42
- remove_nodes_from() (Graph method), 15
- remove_nodes_from() (MultiDiGraph method), 103
- remove_nodes_from() (MultiGraph method), 74
- reverse() (DiGraph method), 68
- reverse() (MultiDiGraph method), 131

S

- `scale_free_graph()` (in module `networkx.generators.directed`), 246
`scipy_discrete_sequence()` (in module `networkx.utils`), 308
`scipy_pareto_sequence()` (in module `networkx.utils`), 307
`scipy_poisson_sequence()` (in module `networkx.utils`), 307
`scipy_powerlaw_sequence()` (in module `networkx.utils`), 307
`scipy_uniform_sequence()` (in module `networkx.utils`), 307
`sedgewick_maze_graph()` (in module `networkx.generators.small`), 225
`selfloop_edges()` (DiGraph method), 65
`selfloop_edges()` (Graph method), 33
`selfloop_edges()` (MultiDiGraph method), 127
`selfloop_edges()` (MultiGraph method), 93
`semantic_feasibility()` (DiGraphMatcher method), 172
`semantic_feasibility()` (GraphMatcher method), 171
`semantic_feasibility()` (WeightedDiGraphMatcher method), 176
`semantic_feasibility()` (WeightedGraphMatcher method), 174
`semantic_feasibility()` (WeightedMultiDiGraphMatcher method), 179
`semantic_feasibility()` (WeightedMultiGraphMatcher method), 177
`shell_layout()` (in module `networkx`), 300
`shortest_path()` (in module `networkx`), 195
`shortest_path_length()` (in module `networkx`), 196
`single_source_dijkstra()` (in module `networkx`), 205
`single_source_dijkstra_path()` (in module `networkx`), 203
`single_source_dijkstra_path_length()` (in module `networkx`), 204
`single_source_shortest_path()` (in module `networkx`), 198
`single_source_shortest_path_length()` (in module `networkx`), 199
`size()` (DiGraph method), 63
`size()` (Graph method), 31
`size()` (MultiDiGraph method), 126
`size()` (MultiGraph method), 92
`spectral_layout()` (in module `networkx`), 301
`spring_layout()` (in module `networkx`), 300
`star_graph()` (in module `networkx.generators.classic`), 221
`stochastic_graph()` (in module `networkx.generators.stochastic`), 252
`strongly_connected_component_subgraphs()` (in module `networkx.algorithms.components.strongly_connected`), 154
`strongly_connected_components()` (in module `networkx.algorithms.components.strongly_connected`), 154
`strongly_connected_components_recursive()` (in module `networkx.algorithms.components.strongly_connected`), 154
`subgraph()` (DiGraph method), 68
`subgraph()` (Graph method), 36
`subgraph()` (MultiDiGraph method), 130
`subgraph()` (MultiGraph method), 96
`subgraph_is_isomorphic()` (DiGraphMatcher method), 172
`subgraph_is_isomorphic()` (GraphMatcher method), 170
`subgraph_is_isomorphic()` (WeightedDiGraphMatcher method), 175
`subgraph_is_isomorphic()` (WeightedGraphMatcher method), 174
`subgraph_is_isomorphic()` (WeightedMultiDiGraphMatcher method), 178
`subgraph_is_isomorphic()` (WeightedMultiGraphMatcher method), 177
`subgraph_isomorphisms_iter()` (DiGraphMatcher method), 172
`subgraph_isomorphisms_iter()` (GraphMatcher method), 170
`subgraph_isomorphisms_iter()` (WeightedDiGraphMatcher method), 176
`subgraph_isomorphisms_iter()` (WeightedGraphMatcher method), 174
`subgraph_isomorphisms_iter()` (WeightedMultiDiGraphMatcher method), 179
`subgraph_isomorphisms_iter()` (WeightedMultiGraphMatcher method), 177
`successors()` (DiGraph method), 55
`successors()` (MultiDiGraph method), 117
`successors_iter()` (DiGraph method), 55
`successors_iter()` (MultiDiGraph method), 117
`symmetric_difference()` (in module `networkx`), 195
`syntactic_feasibility()` (DiGraphMatcher method), 173
`syntactic_feasibility()` (GraphMatcher method), 171
`syntactic_feasibility()` (WeightedDiGraphMatcher method), 176
`syntactic_feasibility()` (WeightedGraphMatcher method), 174
`syntactic_feasibility()` (WeightedMultiDiGraphMatcher method), 179
`syntactic_feasibility()` (WeightedMultiGraphMatcher method), 177

T

- `tetrahedral_graph()` (in module `networkx.generators.small`), 225
`toagraph()` (in module `networkx`), 294
`to_dict_of_dicts()` (in module `networkx.convert`), 262
`to_dict_of_lists()` (in module `networkx.convert`), 263
`to_directed()` (DiGraph method), 67

to_directed() (Graph method), 35
 to_directed() (MultiDiGraph method), 130
 to_directed() (MultiGraph method), 96
 to_edgelist() (in module networkx.convert), 263
 to_networkx_graph() (in module networkx.convert), 259
 to_numpy_matrix() (in module networkx.convert), 264
 to_pydot() (in module networkx), 297
 to_scipy_sparse_matrix() (in module networkx.convert),
 265
 to_undirected() (DiGraph method), 66
 to_undirected() (Graph method), 35
 to_undirected() (MultiDiGraph method), 129
 to_undirected() (MultiGraph method), 95
 topological_sort() (in module networkx), 159
 topological_sort_recursive() (in module networkx), 160
 transitivity() (in module networkx), 148
 triangles() (in module networkx), 148
 trivial_graph() (in module networkx.generators.classic),
 221
 truncated_cube_graph() (in module net-
 workx.generators.small), 225
 truncated_tetrahedron_graph() (in module net-
 workx.generators.small), 225
 tutte_graph() (in module networkx.generators.small), 225

U

uniform_sequence() (in module networkx.utils), 306
 union() (in module networkx), 193
 union() (UnionFind method), 306

W

watts_strogatz_graph() (in module net-
 workx.generators.random_graphs), 231
 weakly_connected_component_subgraphs()
 (in module networkx.algorithms.components.weakly_connected),
 156
 weakly_connected_components() (in module net-
 workx.algorithms.components.weakly_connected),
 156
 wheel_graph() (in module networkx.generators.classic),
 221
 write_adjlist() (in module networkx), 270
 write_dot() (in module networkx), 295, 297
 write_edgelist() (in module networkx), 275
 write_gml() (in module networkx), 278
 write_gpickle() (in module networkx), 280
 write_multiline_adjlist() (in module networkx), 272
 write_pajek() (in module networkx), 283
 write_weighted_edgelist() (in module networkx), 276
 write_yaml() (in module networkx), 281