NetworkX Reference

Release 1.3

Aric Hagberg, Dan Schult, Pieter Swart

August 28, 2010

CONTENTS

CHAPTER

ONE

INTRODUCTION

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

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

1.1 Who uses NetworkX?

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

1.2 The Python programming language

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

1.3 Free software

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

1.4 Goals

NetworkX is intended to:

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

1.5 History

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

1.5.1 What Next

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

CHAPTER

TWO

OVERVIEW

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

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

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

2.1 NetworkX Basics

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

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

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

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

The following basic graph types are provided as Python classes:

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

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

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

MultiDiGraph A directed version of a MultiGraph.

Empty graph-like objects are created with

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

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

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

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

2.1.1 Graphs

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

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

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

2.2 Nodes and Edges

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

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

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

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

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

2.2.1 Graph Creation

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

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

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

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

Edge attributes can be anything:

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

You can add many edges at one time:

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

See the `/tutorial/index` for more examples.

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

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

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

2.2.2 Graph Reporting

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

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

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

2.2.3 Algorithms

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

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

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

2.2.4 Drawing

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

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

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

See the `examples` for more ideas.

2.2.5 Data Structure

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

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

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

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

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

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

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

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

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

CHAPTER

THREE

GRAPH TYPES

NetworkX provides data structures and methods for storing graphs.

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

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

3.1 Which graph class should I use?

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

3.2 Basic graph types

3.2.1 Graph – Undirected graphs with self loops

Overview

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

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

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

Edges:

G can also be grown by adding edges.

Add one edge,

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

a list of edges,

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

or a collection of edges,

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

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

Attributes:

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

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

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

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

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

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

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

Shortcuts:

Many common graph features allow python syntax to speed reporting.

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

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

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

Reporting:

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

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

Adding and removing nodes and edges

Graph.__init__(**attr[, data, name])	Initialize a graph with edges, name, graph attributes.
Graph.add_node(n, **attr[, attr_dict])	Add a single node n and update node attributes.
Graph.add_nodes_from(nodes, **attr)	Add multiple nodes.
Graph.remove_node(n)	Remove node n.
Graph.remove_nodes_from(nodes)	Remove multiple nodes.
Graph.add_edge(u, v, **attr[, attr_dict])	Add an edge between u and v.
Graph.add_edges_from(ebunch, **attr[, attr_dict])	Add all the edges in ebunch.
Graph.add_weighted_edges_from(ebunch, **attr)	Add all the edges in ebunch as weighted edges with specified weights.
Graph.remove_edge(u, v)	Remove the edge between u and v.
Graph.remove_edges_from(ebunch)	Remove all edges specified in ebunch.
Graph.add_star(nlist, **attr)	Add a star.
Graph.add_path(nlist, **attr)	Add a path.
Graph.add_cycle(nlist, **attr)	Add a cycle.
Graph.clear()	Remove all nodes and edges from 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(),key=str)
[0, 1, 2, 'H', 'e', 'l', 'o']
```

Use keywords to update specific node attributes for every node.

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

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

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

networkx.Graph.remove_node

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

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

networkx.Graph.nodes

nodes (*data=False*)

Return a list of the nodes in the graph.

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

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

Returns **nlist** : list

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

Examples

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

networkx.Graph.nodes_iter

nodes_iter (*data=False*)

Return an iterator over the nodes.

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

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

Returns **niter** : iterator

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

Notes

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

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

Examples

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

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

networkx.Graph.__iter__

__iter__()

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

Returns niter : iterator

An iterator over all nodes in the graph.

Examples

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

networkx.Graph.edges

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

Examples

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

networkx.Graph.__getitem__

__getitem__(*n*)

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

Parameters **n** : node

A node in the graph.

Returns **adj_dict** : dictionary

The adjacency dictionary for nodes connected to n.

Notes

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

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

Examples

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

networkx.Graph.adjacency_list

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

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

networkx.Graph.has_node

has_node(*n*)

Return True if the graph contains the node n.

Parameters n : node

Examples

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

It is more readable and simpler to use

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

networkx.Graph.__contains__

__contains__(*n*)

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

Examples

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

networkx.Graph.has_edge

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])
>>> 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])
>>> 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}
>>> list(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)
6.0
```

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

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

networkx.Graph.copy

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

Examples

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

3.2.2 DiGraph - Directed graphs with self loops

Overview

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

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

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

Edges:

G can also be grown by adding edges.

Add one edge,

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

a list of edges,

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

or a collection of edges,

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

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

Attributes:

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

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

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

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

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

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

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

Shortcuts:

Many common graph features allow python syntax to speed reporting.

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

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

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

Reporting:

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

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

Adding and removing nodes and edges

DiGraph.__init__(**attr[, data, name])	Initialize a graph with edges, name, graph attributes.
DiGraph.add_node(n, **attr[, attr_dict])	Add a single node n and update node attributes.
DiGraph.add_nodes_from(nodes, **attr)	Add multiple nodes.
DiGraph.remove_node(n)	Remove node n.
DiGraph.remove_nodes_from(nbunch)	Remove multiple nodes.
DiGraph.add_edge(u, v, **attr[, attr_dict])	Add an edge between u and v.
DiGraph.add_edges_from(ebunch, **attr[, ...])	Add all the edges in ebunch.
DiGraph.add_weighted_edges_from(ebunch, **attr)	Add all the edges in ebunch as weighted edges with specified weights.
DiGraph.remove_edge(u, v)	Remove the edge between u and v.
DiGraph.remove_edges_from(ebunch)	Remove all edges specified in ebunch.
DiGraph.add_star(nlist, **attr)	Add a star.
DiGraph.add_path(nlist, **attr)	Add a path.
DiGraph.add_cycle(nlist, **attr)	Add a cycle.
DiGraph.clear()	Remove all nodes and edges from 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(),key=str)
[0, 1, 2, 'H', 'e', 'l', 'o']
```

Use keywords to update specific node attributes for every node.

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

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

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

networkx.DiGraph.remove_node

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

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

networkx.DiGraph.nodes

nodes (*data=False*)

Return a list of the nodes in the graph.

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

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

Returns **nlist** : list

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

Examples

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

networkx.DiGraph.nodes_iter

nodes_iter(*data=False*)
Return an iterator over the nodes.

Parameters data : boolean, optional (default=False)

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

Returns niter : iterator

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

Notes

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

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

Examples

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

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

networkx.DiGraph.__iter__

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

Returns niter : iterator

An iterator over all nodes in the graph.

Examples

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

networkx.DiGraph.edges

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])
>>> 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

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

networkx.DiGraph.has_node

has_node(*n*)

Return True if the graph contains the node n.

Parameters **n** : node

Examples

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

It is more readable and simpler to use

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

networkx.DiGraph.__contains__

__contains__(*n*)

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

Examples

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

networkx.DiGraph.has_edge

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])
>>> 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])
>>> 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}
>>> list(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}
>>> list(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}
>>> list(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)
6.0
```

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

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

networkx.DiGraph.copy

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

Examples

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

networkx.DiGraph.reverse

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

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

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

Edges:

G can also be grown by adding edges.

Add one edge,

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

a list of edges,

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

or a collection of edges,

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

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

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

Attributes:

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

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

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

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

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

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

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

Shortcuts:

Many common graph features allow python syntax to speed reporting.

```
>>> 1 in G      # check if node in graph
True
>>> [n for n in G if n<3]    # iterate through nodes
[1, 2]
>>> len(G)   # number of nodes in graph
5
>>> G[1] # adjacency dict keyed by neighbor to edge attributes
...         # Note: you should not change this dict manually!
{2: {0: {'weight': 4}, 1: {'color': 'blue'}}}
```

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

```
>>> for n,nbrsdict in G.adjacency_iter():
...     for nbr,keydict in nbrsdict.items():
...        for key,eattr in keydict.items():
...            if 'weight' in eattr:
...                (n,nbr,eattr['weight'])
(1, 2, 4)
```

```
(2, 1, 4)
(2, 3, 8)
(3, 2, 8)
>>> [ (u,v,edata['weight']) for u,v,edata in G.edges(data=True) if 'weight' in edata ]
[(1, 2, 4), (2, 3, 8)]
```

Reporting:

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

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

Adding and removing nodes and edges

`MultiGraph.__init__`(**attr[, data, name])	Initialize a graph with edges, name, graph attributes.
`MultiGraph.add_node`(n, **attr[, attr_dict])	Add a single node n and update node attributes.
`MultiGraph.add_nodes_from`(nodes, **attr)	Add multiple nodes.
`MultiGraph.remove_node`(n)	Remove node n.
`MultiGraph.remove_nodes_from`(nodes)	Remove multiple nodes.
`MultiGraph.add_edge`(u, v, **attr[, key, ...])	Add an edge between u and v.
`MultiGraph.add_edges_from`(ebunch, **attr[, ...])	Add all the edges in ebunch.
`MultiGraph.add_weighted_edges_from`(ebunch, ...)	Add all the edges in ebunch as weighted edges with specified weights.
`MultiGraph.remove_edge`(u, v[, key])	Remove an edge between u and v.
`MultiGraph.remove_edges_from`(ebunch)	Remove all edges specified in ebunch.
`MultiGraph.add_star`(nlist, **attr)	Add a star.
`MultiGraph.add_path`(nlist, **attr)	Add a path.
`MultiGraph.add_cycle`(nlist, **attr)	Add a cycle.
`MultiGraph.clear`()	Remove all nodes and edges from 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(),key=str)
[0, 1, 2, 'H', 'e', 'l', 'o']
```

Use keywords to update specific node attributes for every node.

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

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

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

networkx.MultiGraph.remove_node

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 (abritrary) edge between u and v.

Raises **NetworkXError** :

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

See Also:

remove_edges_from remove a collection of edges

Examples

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

For multiple edges

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

For edges with keys

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

networkx.MultiGraph.remove_edges_from

remove_edges_from(*ebunch*)

Remove all edges specified in ebunch.

Parameters ebunch: list or container of edge tuples :

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

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

See Also:

remove_edge remove a single edge

Notes

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

Examples

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

Removing multiple copies of edges

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

networkx.MultiGraph.add_star

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

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

networkx.MultiGraph.nodes

nodes(*data=False*)

Return a list of the nodes in the graph.

Parameters data : boolean, optional (default=False)

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

Returns nlist : list

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

Examples

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

networkx.MultiGraph.nodes_iter

nodes_iter(*data=False*)

Return an iterator over the nodes.

Parameters data : boolean, optional (default=False)

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

Returns niter : iterator

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

Notes

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

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

Examples

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

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

networkx.MultiGraph.__iter__

__iter__()

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

Returns niter : iterator

An iterator over all nodes in the graph.

Examples

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

networkx.MultiGraph.edges

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

Examples

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

networkx.MultiGraph.__getitem__

__getitem__(*n*)

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

Parameters n : node

A node in the graph.

Returns adj_dict : dictionary

The adjacency dictionary for nodes connected to n.

Notes

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

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

Examples

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

networkx.MultiGraph.adjacency_list

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

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

networkx.MultiGraph.has_node

has_node(*n*)

Return True if the graph contains the node n.

Parameters **n** : node

Examples

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

It is more readable and simpler to use

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

networkx.MultiGraph.__contains__

__contains__(*n*)

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

Examples

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

networkx.MultiGraph.has_edge

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])
>>> 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])
>>> 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}
>>> list(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)
6.0
```

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

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

networkx.MultiGraph.copy

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

Examples

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

3.2.4 MultiDiGraph - Directed graphs with self loops and parallel edges

Overview

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

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

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

Edges:

G can also be grown by adding edges.

Add one edge,

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

a list of edges,

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

or a collection of edges,

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

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

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

Attributes:

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

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

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

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

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

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

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

Shortcuts:

Many common graph features allow python syntax to speed reporting.

```
>>> 1 in G      # check if node in graph
True
>>> [n for n in G if n<3]   # iterate through nodes
[1, 2]
>>> len(G)  # number of nodes in graph
5
>>> G[1] # adjacency dict keyed by neighbor to edge attributes
...      # Note: you should not change this dict manually!
{2: {0: {'weight': 4}, 1: {'color': 'blue'}}}
```

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

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

Reporting:

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

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

Adding and Removing Nodes and Edges

MultiDiGraph.__init__(**attr[, data, name])	Initialize a graph with edges, name, graph attributes.
MultiDiGraph.add_node(n, **attr[, attr_dict])	Add a single node n and update node attributes.
MultiDiGraph.add_nodes_from(nodes, **attr)	Add multiple nodes.
MultiDiGraph.remove_node(n)	Remove node n.
MultiDiGraph.remove_nodes_from(nbunch)	Remove multiple nodes.
MultiDiGraph.add_edge(u, v, **attr[, key, ...])	Add an edge between u and v.
MultiDiGraph.add_edges_from(ebunch, **attr)	Add all the edges in ebunch.
MultiDiGraph.add_weighted_edges_from(ebunch, ...)	Add all the edges in ebunch as weighted edges with specified weights.
MultiDiGraph.remove_edge(u, v[, key])	Remove an edge between u and v.
MultiDiGraph.remove_edges_from(ebunch)	Remove all edges specified in ebunch.
MultiDiGraph.add_star(nlist, **attr)	Add a star.
MultiDiGraph.add_path(nlist, **attr)	Add a path.
MultiDiGraph.add_cycle(nlist, **attr)	Add a cycle.
MultiDiGraph.clear()	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(),key=str)
[0, 1, 2, 'H', 'e', 'l', 'o']
```

Use keywords to update specific node attributes for every node.

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

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

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

networkx.MultiDiGraph.remove_node

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 (abritrary) edge between u and v.

Raises NetworkXError :

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

See Also:

remove_edges_from remove a collection of edges

Examples

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

For multiple edges

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

For edges with keys

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

networkx.MultiDiGraph.remove_edges_from

remove_edges_from (*ebunch*)

Remove all edges specified in ebunch.

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

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

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

See Also:

`remove_edge` remove a single edge

Notes

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

Examples

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

Removing multiple copies of edges

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

networkx.MultiDiGraph.add_star

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

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

networkx.MultiDiGraph.nodes

nodes(*data=False*)

Return a list of the nodes in the graph.

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

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

Returns **nlist** : list

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

Examples

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

networkx.MultiDiGraph.nodes_iter

nodes_iter(*data=False*)

Return an iterator over the nodes.

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

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

Returns **niter** : iterator

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

Notes

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

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

Examples

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

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

networkx.MultiDiGraph.__iter__

__iter__()

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

Returns niter : iterator

An iterator over all nodes in the graph.

Examples

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

networkx.MultiDiGraph.edges

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])
>>> 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

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

networkx.MultiDiGraph.has_node

has_node(*n*)

Return True if the graph contains the node n.

Parameters **n** : node

Examples

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

It is more readable and simpler to use

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

networkx.MultiDiGraph.__contains__

__contains__(*n*)

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

Examples

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

networkx.MultiDiGraph.has_edge

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])
>>> 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])
>>> 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}
>>> list(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}
>>> list(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}
>>> list(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)
6.0
```

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

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

networkx.MultiDiGraph.copy

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

Examples

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

networkx.MultiDiGraph.reverse

reverse(*copy=True*)

Return the reverse of the graph.

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

Parameters copy : bool optional (default=True)

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

CHAPTER

FOUR

ALGORITHMS

4.1 Bipartite

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

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)
>>> list(X)
[0, 2]
>>> list(Y)
[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>

blockmodel(G, partitions[, multigraph])	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

[R49]

Examples

```
>>> G=nx.path_graph(6)
>>> partition=[[0,1],[2,3],[4,5]]
>>> M=nx.blockmodel(G,partition)
```

4.3 Boundary

Routines to find the boundary of a set of nodes.

Edge boundaries are edges that have only one end in the set of nodes.

Node boundaries are nodes outside the set of nodes that have an edge to a node in the set.

`edge_boundary`(G, nbunch1[, nbunch2])	Return the edge boundary.
`node_boundary`(G, nbunch1[, nbunch2])	Return the node boundary.

4.3.1 networkx.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

Degree centrality measures.

degree_centrality(G)	Compute the degree centrality for nodes.
in_degree_centrality(G)	Compute the in-degree centrality for nodes.
out_degree_centrality(G)	Compute the out-degree centrality for nodes.

networkx.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 compoenent containing v.

Returns nodes : dictionary

Dictionary of nodes with closeness centrality as the value.

See Also:

`betweenness_centrality`, `load_centrality`, `eigenvector_centrality`, `degree_centrality`

Notes

The closeness centrality is normalized to to n-1 / size(G)-1 where n is the number of nodes in the connected part of graph containing the node. If the graph is not completely connected, this algorithm computes the closeness centrality for each connected part separately.

4.4.3 Betweenness

Betweenness centrality measures.

`betweenness_centrality`(G[, normalized, ...])	Compute betweenness centrality for nodes.
`edge_betweenness_centrality`(G[, normalized, ...])	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, 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 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 [R38].

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

[R38]

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 [R39].

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

[R39]

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 [R50].

If the edges have a 'weight' attribute they will be used as weights in this algorithm. Unspecified weights are set to 1.

See also [R51] for the original definition of information centrality.

References

[R50], [R51]

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 [R41].

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 [R40].

If the edges have a 'weight' attribute they will be used as weights in this algorithm. Unspecified weights are set to 1.

References

[R40], [R41]

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 [R43].

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 [R42].

If the edges have a 'weight' attribute they will be used as weights in this algorithm. Unspecified weights are set to 1.

References

[R42], [R43]

4.4.6 Eigenvector

Eigenvector centrality.

eigenvector_centrality(G[, max_iter, tol, ...])	Compute the eigenvector centrality for the graph G.
eigenvector_centrality_numpy(G)	Compute the eigenvector centrality for the graph G.

networkx.eigenvector_centrality

eigenvector_centrality(*G, max_iter=100, tol=9.9999999999999995e-07, nstart=None*)

Compute the eigenvector centrality for the graph G.

Uses the power method to find the eigenvector for the largest eigenvalue of the adjacency matrix of G.

Parameters **G** : graph

A networkx graph

max_iter : interger, optional

Maximum number of iterations in power method.

tol : float, optional

Error tolerance used to check convergence in power method iteration.

nstart : dictionary, optional

Starting value of eigenvector iteration for each node.

Returns **nodes** : dictionary

Dictionary of nodes with eigenvector centrality as the value.

See Also:

eigenvector_centrality_numpy, pagerank, hits

Notes

The eigenvector calculation is done by the power iteration method and has no guarantee of convergence. The iteration will stop after max_iter iterations or an error tolerance of number_of_nodes(G)*tol has been reached.

For directed graphs this is “right” eigevector centrality. For “left” eigenvector centrality, first reverse the graph with G.reverse().

Examples

```
>>> G=nx.path_graph(4)
>>> centrality=nx.eigenvector_centrality(G)
>>> print(['%s %0.2f'%(node,centrality[node]) for node in centrality])
['0 0.37', '1 0.60', '2 0.60', '3 0.37']
```

networkx.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” eigevector centrality. For “left” eigenvector centrality, first reverse the graph with G.reverse().

Examples

```
>>> G=nx.path_graph(4)
>>> centrality=nx.eigenvector_centrality_numpy(G)
>>> print(['%s %0.2f'%(node,centrality[node]) for node in centrality])
['0 0.37', '1 0.60', '2 0.60', '3 0.37']
```

4.4.7 Load

Load centrality.

load_centrality(G[, v, cutoff, normalized, ...])	Compute load centrality for nodes.
edge_load(G[, nodes, cutoff])	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, optional

If True the betweenness values are normalized by b=b/(n-1)(n-2) where n is the number of nodes in G.

weighted_edges : bool, optional

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

cutoff : bool, optional

If specified, only consider paths of length <= cutoff.

Returns nodes : dictionary

Dictionary of nodes with centrality as the value.

See Also:

`betweenness_centrality`

Notes

Load centrality is slightly different than betweenness. For this load algorithm see the reference Scientific collaboration networks: II. Shortest paths, weighted networks, and centrality, M. E. J. Newman, Phys. Rev. E 64, 016132 (2001).

networkx.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

find_cliques(G)	Search for all maximal cliques in a graph.
make_max_clique_graph(G[, create_using, name])	Create the maximal clique graph of a graph.
make_clique_bipartite(G[, fpos, ...])	Create a bipartite clique graph from a graph G.
graph_clique_number(G[, cliques])	Return the clique number (size of the largest clique) for G.
graph_number_of_cliques(G[, cliques])	Returns the number of maximal cliques in G.
node_clique_number(G[, nodes, cliques])	Returns the size of the largest maximal clique containing each given node.
number_of_cliques(G[, nodes, cliques])	Returns the number of maximal cliques for each node.
cliques_containing_node(G[, nodes, cliques])	Returns a list of cliques containing the given node.

4.5.1 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`

Notes

Based on the algorithm published by Bron & Kerbosch (1973) [R56] as adapated by Tomita, Tanaka and Takahashi (2006) [R57] and discussed in Cazals and Karande (2008) [R58].

References

[R56], [R57], [R58]

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.

triangles(G[, nbunch])	Compute the number of triangles.
transitivity(G)	Compute transitivity.
clustering(G[, nbunch, weights])	Compute the clustering coefficient for nodes.
average_clustering(G)	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 trianges keyed by node label.

Notes

When computing triangles for the entire graph each triangle is counted three times, once at each node.

Self loops are ignored.

Examples

```
>>> G=nx.complete_graph(5)
>>> print(nx.triangles(G,0))
6
>>> print(nx.triangles(G))
{0: 6, 1: 6, 2: 6, 3: 6, 4: 6}
>>> print(list(nx.triangles(G,(0,1)).values()))
[6, 6]
```

4.6.2 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*triangles/triads

Parameters G : graph

A networkx graph

Returns **out** : float

Transitivity

Examples

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

4.6.3 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. Ths is useful for computing transitivity.

Self loops are ignored.

Examples

```
>>> G=nx.complete_graph(5)
>>> print(nx.clustering(G,0))
1.0
>>> print(nx.clustering(G))
{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.

is_connected(G)	Test graph connectivity.
number_connected_components(G)	Return number of connected components in graph.
connected_components(G)	Return nodes in connected components of graph.
connected_component_subgraphs(G)	Return connected components as subgraphs.
node_connected_component(G, n)	Return nodes in connected components of graph containing node n.

networkx.algorithms.components.connected.is_connected

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.

is_strongly_connected(G)	Test directed graph for strong connectivity.
number_strongly_connected_components(G)	Return number of strongly connected components in graph.
strongly_connected_components(G)	Return nodes in strongly connected components of graph.
strongly_connected_component_subgraphs(G)	Return strongly connected components as subgraphs.
strongly_connected_components_recursive(G)	Return nodes in strongly connected components of graph.
kosaraju_strongly_connected_components(G[, ...])	Return nodes in strongly connected components of graph.
condensation(G)	Returns the condensation of G.

networkx.algorithms.components.strongly_connected.is_strongly_connected

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

[R44], [R45]

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

[R46], [R47]

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.

`is_weakly_connected`(G)	Test directed graph for weak connectivity.
`number_weakly_connected_components`(G)	Return the number of connected components in G.
`weakly_connected_components`(G)	Return weakly connected components of G.
`weakly_connected_component_subgraphs`(G)	Return weakly connected components as subgraphs.

networkx.algorithms.components.weakly_connected.is_weakly_connected

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 Atrracting components

Attracting components.

is_attracting_component(G)	Returns True if *G* consists of a single attracting component.
number_attracting_components(G)	Returns the number of attracting components in *G*.
attracting_components(G)	Returns a list of attracting components in *G*.
attracting_component_subgraphs(G)	Returns a list of attracting component subgraphs from *G*.

networkx.algorithms.components.attracting.is_attracting_component

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 ditionary keyed by node to the core number.

References

[R59]

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).

topological_sort(G[, nbunch])	Return a list of nodes in topological sort order.
topological_sort_recursive(G[, nbunch])	Return a list of nodes in topological sort order.
is_directed_acyclic_graph(G)	Return True if the graph G is a directed acyclic graph (DAG) or False if not.

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

Raises **NetworkXError** :

Topological sort is defined for directed graphs only. If the graph G is undirected, a NetworkXError is raised.

NetworkXUnfeasible :

If G is not a directed acyclic graph (DAG) no topological sort exists and a NetworkX-Unfeasible exception is raised.

See Also:

`is_directed_acyclic_graph`

Notes

This algorithm is based on a description and proof in The Algorithm Design Manual [R105] .

References

[R105]

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

Raises **NetworkXError** :

Topological sort is defined for directed graphs only. If the graph G is undirected, a NetworkXError is raised.

NetworkXUnfeasible :

If G is not a directed acyclic graph (DAG) no topological sort exists and a NetworkX-Unfeasible exception is raised.

See Also:

topological_sort, is_directed_acyclic_graph

Notes

This is a recursive version of topological sort.

4.10.3 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.

center(G[, e])	Return the periphery of the graph G.
diameter(G[, e])	Return the diameter of the graph G.
eccentricity(G[, v, sp])	Return the eccentricity of nodes in G.
periphery(G[, e])	Return the periphery of the graph G.
radius(G[, e])	Return the radius of the graph G.

4.11.1 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 lenghts 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.

is_eulerian(G)	Return True if G is an Eulerian graph, False otherwise.
eulerian_circuit(G[, source])	Return the edges of an Eulerian circuit in G.

4.12.1 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

```
>>> nx.is_eulerian(nx.DiGraph({0:[3], 1:[2], 2:[3], 3:[0, 1]}))
True
>>> nx.is_eulerian(nx.complete_graph(5))
True
>>> nx.is_eulerian(nx.petersen_graph())
False
```

4.12.2 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 [R54],[R55]_

References

[R54], [R55]

Examples

```
>>> G=nx.complete_graph(3)
>>> list(nx.eulerian_circuit(G))
[(0, 1), (1, 2), (2, 0)]
>>> list(nx.eulerian_circuit(G,source=1))
[(1, 0), (0, 2), (2, 1)]
>>> [u for u,v in nx.eulerian_circuit(G)]  # nodes in circuit
[0, 1, 2]
```

4.13 Flows

4.13.1 Ford-Fulkerson

max_flow(G, s, t[, capacity])	Find the value of a maximum single-commodity flow.
min_cut(G, s, t[, capacity])	Compute the value of a minimum (s, t)-cut.
ford_fulkerson(G, s, t[, capacity])	Find a maximum single-commodity flow using the Ford-Fulkerson algorithm.
ford_fulkerson_flow(G, s, t[, capacity])	Return a maximum flow for a single-commodity flow problem.

networkx.max_flow

max_flow (*G, s, t, capacity='capacity'*)

Find the value of a maximum single-commodity flow.

Parameters **G** : NetworkX graph

Edges of the graph are expected to have an attribute called 'capacity'. If this attribute is not present, the edge is considered to have infinite capacity.

s : node

Source node for the flow.

t : node

Sink node for the flow.

capacity: string :

Edges of the graph G are expected to have an attribute capacity that indicates how much flow the edge can support. If this attribute is not present, the edge is considered to have infinite capacity. Default value: 'capacity'.

Returns flowValue : integer, float

Value of the maximum flow, i.e., net outflow from the source.

Raises NetworkXError :

The algorithm does not support MultiGraph and MultiDiGraph. If the input graph is an instance of one of these two classes, a NetworkXError is raised.

NetworkXUnbounded :

If the graph has a path of infinite capacity, the value of a feasible flow on the graph is unbounded above and the function raises a NetworkXUnbounded.

Examples

```
>>> import networkx as nx
>>> G = nx.DiGraph()
>>> G.add_edge('x','a', capacity = 3.0)
>>> G.add_edge('x','b', capacity = 1.0)
>>> G.add_edge('a','c', capacity = 3.0)
>>> G.add_edge('b','c', capacity = 5.0)
>>> G.add_edge('b','d', capacity = 4.0)
>>> G.add_edge('d','e', capacity = 2.0)
>>> G.add_edge('c','y', capacity = 2.0)
>>> G.add_edge('e','y', capacity = 3.0)
>>> flow=nx.max_flow(G, 'x', 'y')
>>> flow
3.0
```

networkx.min_cut

min_cut (*G, s, t, capacity='capacity'*)

Compute the value of a minimum (s, t)-cut.

Use the max-flow min-cut theorem, i.e., the capacity of a minimum capacity cut is equal to the flow value of a maximum flow.

Parameters G : NetworkX graph

Edges of the graph are expected to have an attribute called 'capacity'. If this attribute is not present, the edge is considered to have infinite capacity.

s : node

Source node for the flow.

t : node

Sink node for the flow.

capacity: string :

Edges of the graph G are expected to have an attribute capacity that indicates how much flow the edge can support. If this attribute is not present, the edge is considered to have infinite capacity. Default value: 'capacity'.

Returns cutValue : integer, float

Value of the minimum cut.

Raises NetworkXUnbounded :

If the graph has a path of infinite capacity, all cuts have infinite capacity and the function raises a NetworkXError.

Examples

```
>>> import networkx as nx
>>> G = nx.DiGraph()
>>> G.add_edge('x','a', capacity = 3.0)
>>> G.add_edge('x','b', capacity = 1.0)
>>> G.add_edge('a','c', capacity = 3.0)
>>> G.add_edge('b','c', capacity = 5.0)
>>> G.add_edge('b','d', capacity = 4.0)
>>> G.add_edge('d','e', capacity = 2.0)
>>> G.add_edge('c','y', capacity = 2.0)
>>> G.add_edge('e','y', capacity = 3.0)
>>> nx.min_cut(G, 'x', 'y')
3.0
```

networkx.ford_fulkerson

ford_fulkerson(*G, s, t, capacity='capacity'*)

Find a maximum single-commodity flow using the Ford-Fulkerson algorithm.

This algorithm uses Edmonds-Karp-Dinitz path selection rule which guarantees a running time of O(nm^2) for n nodes and m edges.

Parameters G : NetworkX graph

Edges of the graph are expected to have an attribute called 'capacity'. If this attribute is not present, the edge is considered to have infinite capacity.

s : node

Source node for the flow.

t : node

Sink node for the flow.

capacity: string :

Edges of the graph G are expected to have an attribute capacity that indicates how much flow the edge can support. If this attribute is not present, the edge is considered to have infinite capacity. Default value: 'capacity'.

Returns flowValue : integer, float

Value of the maximum flow, i.e., net outflow from the source.

flowDict : dictionary

Dictionary of dictionaries keyed by nodes such that flowDict[u][v] is the flow edge (u, v).

Raises NetworkXError :

The algorithm does not support MultiGraph and MultiDiGraph. If the input graph is an instance of one of these two classes, a NetworkXError is raised.

NetworkXUnbounded :

If the graph has a path of infinite capacity, the value of a feasible flow on the graph is unbounded above and the function raises a NetworkXUnbounded.

Examples

```
>>> import networkx as nx
>>> G = nx.DiGraph()
>>> G.add_edge('x','a', capacity = 3.0)
>>> G.add_edge('x','b', capacity = 1.0)
>>> G.add_edge('a','c', capacity = 3.0)
>>> G.add_edge('b','c', capacity = 5.0)
>>> G.add_edge('b','d', capacity = 4.0)
>>> G.add_edge('d','e', capacity = 2.0)
>>> G.add_edge('c','y', capacity = 2.0)
>>> G.add_edge('e','y', capacity = 3.0)
>>> flow,F=nx.ford_fulkerson(G, 'x', 'y')
>>> flow
3.0
```

networkx.ford_fulkerson_flow

ford_fulkerson_flow(*G, s, t, capacity='capacity'*)

Return a maximum flow for a single-commodity flow problem.

Parameters G : NetworkX graph

Edges of the graph are expected to have an attribute called 'capacity'. If this attribute is not present, the edge is considered to have infinite capacity.

s : node

Source node for the flow.

t : node

Sink node for the flow.

capacity: string :

Edges of the graph G are expected to have an attribute capacity that indicates how much flow the edge can support. If this attribute is not present, the edge is considered to have infinite capacity. Default value: 'capacity'.

Returns flowDict : dictionary

Dictionary of dictionaries keyed by nodes such that flowDict[u][v] is the flow edge (u, v).

Raises NetworkXError :

The algorithm does not support MultiGraph and MultiDiGraph. If the input graph is an instance of one of these two classes, a NetworkXError is raised.

NetworkXUnbounded :

If the graph has a path of infinite capacity, the value of a feasible flow on the graph is unbounded above and the function raises a NetworkXUnbounded.

Examples

```
>>> import networkx as nx
>>> G = nx.DiGraph()
>>> G.add_edge('x','a', capacity = 3.0)
>>> G.add_edge('x','b', capacity = 1.0)
>>> G.add_edge('a','c', capacity = 3.0)
>>> G.add_edge('b','c', capacity = 5.0)
>>> G.add_edge('b','d', capacity = 4.0)
>>> G.add_edge('d','e', capacity = 2.0)
>>> G.add_edge('c','y', capacity = 2.0)
>>> G.add_edge('e','y', capacity = 3.0)
>>> F=nx.ford_fulkerson_flow(G, 'x', 'y')
>>> for u, v in G.edges_iter():
...     print('(%s, %s) %.2f' % (u, v, F[u][v]))
...
(a, c) 2.00
(c, y) 2.00
(b, c) 0.00
(b, d) 1.00
(e, y) 1.00
(d, e) 1.00
(x, a) 2.00
(x, b) 1.00
```

4.13.2 Network Simplex

network_simplex(G[, demand, capacity, weight])	Find a minimum cost flow satisfying all demands in digraph G.
min_cost_flow_cost(G[, demand, capacity, weight])	Find the cost of a minimum cost flow satisfying all demands in digraph G.
min_cost_flow(G[, demand, capacity, weight])	Return a minimum cost flow satisfying all demands in digraph G.
cost_of_flow(G, flowDict[, weight])	Compute the cost of the flow given by flowDict on graph G.
max_flow_min_cost(G, s, t[, capacity, weight])	Return a maximum (s, t)-flow of minimum cost.

networkx.network_simplex

network_simplex (*G, demand='demand', capacity='capacity', weight='weight'*)

Find a minimum cost flow satisfying all demands in digraph G.

This is a primal network simplex algorithm that uses the leaving arc rule to prevent cycling.

G is a digraph with edge costs and capacities and in which nodes have demand, i.e., they want to send or receive some amount of flow. A negative demand means that the node wants to send flow, a positive demand means that the node want to receive flow. A flow on the digraph G satisfies all demand if the net flow into each node is equal to the demand of that node.

Parameters **G** : NetworkX graph

DiGraph on which a minimum cost flow satisfying all demands is to be found.

demand: string :

Nodes of the graph G are expected to have an attribute demand that indicates how much flow a node wants to send (negative demand) or receive (positive demand). Note that the sum of the demands should be 0 otherwise the problem in not feasible. If this attribute is not present, a node is considered to have 0 demand. Default value: 'demand'.

capacity: string :

Edges of the graph G are expected to have an attribute capacity that indicates how much flow the edge can support. If this attribute is not present, the edge is considered to have infinite capacity. Default value: 'capacity'.

weight: string :

Edges of the graph G are expected to have an attribute weight that indicates the cost incurred by sending one unit of flow on that edge. If not present, the weight is considered to be 0. Default value: 'weight'.

Returns **flowCost: integer, float** :

Cost of a minimum cost flow satisfying all demands.

flowDict: dictionary :

Dictionary of dictionaries keyed by nodes such that flowDict[u][v] is the flow edge (u, v).

Raises **NetworkXError** :

This exception is raised if the input graph is not directed or not connected.

NetworkXUnfeasible :

This exception is raised in the following situations:

- The sum of the demands is not zero. Then, there is no flow satisfying all demands.
- There is no flow satisfying all demand.

NetworkXUnbounded :

This exception is raised if the digraph G has a cycle of negative cost and infinite capacity. Then, the cost of a flow satisfying all demands is unbounded below.

See Also:

`cost_of_flow`, `max_flow_min_cost`, `min_cost_flow`, `min_cost_flow_cost`

References

W. J. Cook, W. H. Cunningham, W. R. Pulleyblank and A. Schrijver. Combinatorial Optimization. Wiley-Interscience, 1998.

Examples

A simple example of a min cost flow problem.

```
>>> import networkx as nx
>>> G = nx.DiGraph()
>>> G.add_node('a', demand = -5)
>>> G.add_node('d', demand = 5)
>>> G.add_edge('a', 'b', weight = 3, capacity = 4)
>>> G.add_edge('a', 'c', weight = 6, capacity = 10)
>>> G.add_edge('b', 'd', weight = 1, capacity = 9)
>>> G.add_edge('c', 'd', weight = 2, capacity = 5)
>>> flowCost, flowDict = nx.network_simplex(G)
>>> flowCost
24
>>> flowDict
{'a': {'c': 1, 'b': 4}, 'c': {'d': 1}, 'b': {'d': 4}, 'd': {}}
```

The mincost flow algorithm can also be used to solve shortest path problems. To find the shortest path between two nodes u and v, give all edges an infinite capacity, give node u a demand of -1 and node v a demand a 1. Then run the network simplex. The value of a min cost flow will be the distance between u and v and edges carrying positive flow will indicate the path.

```
>>> G=nx.DiGraph()
>>> G.add_weighted_edges_from([('s','u',10), ('s','x',5),
...                            ('u','v',1), ('u','x',2),
...                            ('v','y',1), ('x','u',3),
...                            ('x','v',5), ('x','y',2),
...                            ('y','s',7), ('y','v',6)])
>>> G.add_node('s', demand = -1)
>>> G.add_node('v', demand = 1)
>>> flowCost, flowDict = nx.network_simplex(G)
>>> flowCost == nx.shortest_path_length(G, 's', 'v', weighted = True)
True
>>> [(u, v) for u in flowDict for v in flowDict[u] if flowDict[u][v] > 0]
[('x', 'u'), ('s', 'x'), ('u', 'v')]
>>> nx.shortest_path(G, 's', 'v', weighted = True)
['s', 'x', 'u', 'v']
```

It is possible to change the name of the attributes used for the algorithm.

```
>>> G = nx.DiGraph()
>>> G.add_node('p', spam = -4)
>>> G.add_node('q', spam = 2)
>>> G.add_node('a', spam = -2)
>>> G.add_node('d', spam = -1)
>>> G.add_node('t', spam = 2)
>>> G.add_node('w', spam = 3)
>>> G.add_edge('p', 'q', cost = 7, vacancies = 5)
>>> G.add_edge('p', 'a', cost = 1, vacancies = 4)
>>> G.add_edge('q', 'd', cost = 2, vacancies = 3)
>>> G.add_edge('t', 'q', cost = 1, vacancies = 2)
>>> G.add_edge('a', 't', cost = 2, vacancies = 4)
>>> G.add_edge('d', 'w', cost = 3, vacancies = 4)
>>> G.add_edge('t', 'w', cost = 4, vacancies = 1)
>>> flowCost, flowDict = nx.network_simplex(G, demand = 'spam',
...                                         capacity = 'vacancies',
```

```
...                                           weight = 'cost')
>>> flowCost
37
>>> flowDict
{'a': {'t': 4}, 'd': {'w': 2}, 'q': {'d': 1}, 'p': {'q': 2, 'a': 2}, 't': {'q': 1, 'w': 1}, 'w':
```

networkx.min_cost_flow_cost

min_cost_flow_cost (*G, demand='demand', capacity='capacity', weight='weight'*)

Find the cost of a minimum cost flow satisfying all demands in digraph G.

G is a digraph with edge costs and capacities and in which nodes have demand, i.e., they want to send or receive some amount of flow. A negative demand means that the node wants to send flow, a positive demand means that the node want to receive flow. A flow on the digraph G satisfies all demand if the net flow into each node is equal to the demand of that node.

Parameters **G** : NetworkX graph

DiGraph on which a minimum cost flow satisfying all demands is to be found.

demand: string :

Nodes of the graph G are expected to have an attribute demand that indicates how much flow a node wants to send (negative demand) or receive (positive demand). Note that the sum of the demands should be 0 otherwise the problem in not feasible. If this attribute is not present, a node is considered to have 0 demand. Default value: 'demand'.

capacity: string :

Edges of the graph G are expected to have an attribute capacity that indicates how much flow the edge can support. If this attribute is not present, the edge is considered to have infinite capacity. Default value: 'capacity'.

weight: string :

Edges of the graph G are expected to have an attribute weight that indicates the cost incurred by sending one unit of flow on that edge. If not present, the weight is considered to be 0. Default value: 'weight'.

Returns **flowCost: integer, float** :

Cost of a minimum cost flow satisfying all demands.

Raises **NetworkXError** :

This exception is raised if the input graph is not directed or not connected.

NetworkXUnfeasible :

This exception is raised in the following situations:

- The sum of the demands is not zero. Then, there is no flow satisfying all demands.
- There is no flow satisfying all demand.

NetworkXUnbounded :

This exception is raised if the digraph G has a cycle of negative cost and infinite capacity. Then, the cost of a flow satisfying all demands is unbounded below.

See Also:

cost_of_flow, max_flow_min_cost, min_cost_flow, network_simplex

Examples

A simple example of a min cost flow problem.

```
>>> import networkx as nx
>>> G = nx.DiGraph()
>>> G.add_node('a', demand = -5)
>>> G.add_node('d', demand = 5)
>>> G.add_edge('a', 'b', weight = 3, capacity = 4)
>>> G.add_edge('a', 'c', weight = 6, capacity = 10)
>>> G.add_edge('b', 'd', weight = 1, capacity = 9)
>>> G.add_edge('c', 'd', weight = 2, capacity = 5)
>>> flowCost = nx.min_cost_flow_cost(G)
>>> flowCost
24
```

networkx.min_cost_flow

min_cost_flow(*G, demand='demand', capacity='capacity', weight='weight'*)

Return a minimum cost flow satisfying all demands in digraph G.

G is a digraph with edge costs and capacities and in which nodes have demand, i.e., they want to send or receive some amount of flow. A negative demand means that the node wants to send flow, a positive demand means that the node want to receive flow. A flow on the digraph G satisfies all demand if the net flow into each node is equal to the demand of that node.

Parameters **G** : NetworkX graph

DiGraph on which a minimum cost flow satisfying all demands is to be found.

demand: string :

Nodes of the graph G are expected to have an attribute demand that indicates how much flow a node wants to send (negative demand) or receive (positive demand). Note that the sum of the demands should be 0 otherwise the problem in not feasible. If this attribute is not present, a node is considered to have 0 demand. Default value: 'demand'.

capacity: string :

Edges of the graph G are expected to have an attribute capacity that indicates how much flow the edge can support. If this attribute is not present, the edge is considered to have infinite capacity. Default value: 'capacity'.

weight: string :

Edges of the graph G are expected to have an attribute weight that indicates the cost incurred by sending one unit of flow on that edge. If not present, the weight is considered to be 0. Default value: 'weight'.

Returns **flowDict: dictionary** :

Dictionary of dictionaries keyed by nodes such that flowDict[u][v] is the flow edge (u, v).

Raises **NetworkXError** :

This exception is raised if the input graph is not directed or not connected.

NetworkXUnfeasible :

This exception is raised in the following situations:

- The sum of the demands is not zero. Then, there is no flow satisfying all demands.
- There is no flow satisfying all demand.

NetworkXUnbounded :

This exception is raised if the digraph G has a cycle of negative cost and infinite capacity. Then, the cost of a flow satisfying all demands is unbounded below.

See Also:

cost_of_flow, max_flow_min_cost, min_cost_flow_cost, network_simplex

Examples

A simple example of a min cost flow problem.

```
>>> import networkx as nx
>>> G = nx.DiGraph()
>>> G.add_node('a', demand = -5)
>>> G.add_node('d', demand = 5)
>>> G.add_edge('a', 'b', weight = 3, capacity = 4)
>>> G.add_edge('a', 'c', weight = 6, capacity = 10)
>>> G.add_edge('b', 'd', weight = 1, capacity = 9)
>>> G.add_edge('c', 'd', weight = 2, capacity = 5)
>>> flowDict = nx.min_cost_flow(G)
>>> flowDict
{'a': {'c': 1, 'b': 4}, 'c': {'d': 1}, 'b': {'d': 4}, 'd': {}}
```

networkx.cost_of_flow

cost_of_flow (*G, flowDict, weight='weight'*)

Compute the cost of the flow given by flowDict on graph G.

Note that this function does not check for the validity of the flow flowDict. This function will fail if the graph G and the flow don't have the same edge set.

Parameters G : NetworkX graph

DiGraph on which a minimum cost flow satisfying all demands is to be found.

weight: string :

Edges of the graph G are expected to have an attribute weight that indicates the cost incurred by sending one unit of flow on that edge. If not present, the weight is considered to be 0. Default value: 'weight'.

flowDict: dictionary :

Dictionary of dictionaries keyed by nodes such that flowDict[u][v] is the flow edge (u, v).

Returns cost: Integer, float :

The total cost of the flow. This is given by the sum over all edges of the product of the edge's flow and the edge's weight.

See Also:

max_flow_min_cost, min_cost_flow, min_cost_flow_cost, network_simplex

networkx.max_flow_min_cost

max_flow_min_cost (*G, s, t, capacity='capacity', weight='weight'*)

Return a maximum (s, t)-flow of minimum cost.

G is a digraph with edge costs and capacities. There is a source node s and a sink node t. This function finds a maximum flow from s to t whose total cost is minimized.

Parameters **G** : NetworkX graph

DiGraph on which a minimum cost flow satisfying all demands is to be found.

s: node label :

Source of the flow.

t: node label :

Destination of the flow.

capacity: string :

Edges of the graph G are expected to have an attribute capacity that indicates how much flow the edge can support. If this attribute is not present, the edge is considered to have infinite capacity. Default value: 'capacity'.

weight: string :

Edges of the graph G are expected to have an attribute weight that indicates the cost incurred by sending one unit of flow on that edge. If not present, the weight is considered to be 0. Default value: 'weight'.

Returns **flowDict: dictionary** :

Dictionary of dictionaries keyed by nodes such that flowDict[u][v] is the flow edge (u, v).

Raises **NetworkXError** :

This exception is raised if the input graph is not directed or not connected.

NetworkXUnbounded :

This exception is raised if there is an infinite capacity path from s to t in G. In this case there is no maximum flow. This exception is also raised if the digraph G has a cycle of negative cost and infinite capacity. Then, the cost of a flow is unbounded below.

See Also:

`cost_of_flow`, `ford_fulkerson`, `min_cost_flow`, `min_cost_flow_cost`, `network_simplex`

Examples

```
>>> G = nx.DiGraph()
>>> G.add_edges_from([(1, 2, {'capacity': 12, 'weight': 4}),
...                   (1, 3, {'capacity': 20, 'weight': 6}),
...                   (2, 3, {'capacity': 6, 'weight': -3}),
...                   (2, 6, {'capacity': 14, 'weight': 1}),
...                   (3, 4, {'weight': 9}),
...                   (3, 5, {'capacity': 10, 'weight': 5}),
...                   (4, 2, {'capacity': 19, 'weight': 13}),
```

```
...                    (4, 5, {'capacity': 4, 'weight': 0}),
...                    (5, 7, {'capacity': 28, 'weight': 2}),
...                    (6, 5, {'capacity': 11, 'weight': 1}),
...                    (6, 7, {'weight': 8}),
...                    (7, 4, {'capacity': 6, 'weight': 6})])
>>> mincostFlow = nx.max_flow_min_cost(G, 1, 7)
>>> nx.cost_of_flow(G, mincostFlow)
373
>>> maxFlow = nx.ford_fulkerson_flow(G, 1, 7)
>>> nx.cost_of_flow(G, maxFlow)
428
>>> mincostFlowValue = (sum((mincostFlow[u][7] for u in G.predecessors(7)))
...                     - sum((mincostFlow[7][v] for v in G.successors(7))))
>>> mincostFlowValue == nx.max_flow(G, 1, 7)
True
```

4.14 Isolates

Functions for identifying isolate (degree zero) nodes.

is_isolate(G, n)	Determine of node n is an isolate (degree zero).
isolates(G)	Return list of isolates in the graph.

4.14.1 networkx.is_isolate

is_isolate (*G*, *n*)

Determine of node n is an isolate (degree zero).

Parameters **G** : graph

A networkx graph

n : node

A node in G

Returns **isolate** : bool

True if n has no neighbors, False otherwise.

Examples

```
>>> G=nx.Graph()
>>> G.add_edge(1,2)
>>> G.add_node(3)
>>> nx.is_isolate(G,2)
False
>>> nx.is_isolate(G,3)
True
```

4.14.2 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

`is_isomorphic`(G1, G2[, weighted, rtol, atol])	Returns True if the graphs G1 and G2 are isomorphic and False otherwise.
`could_be_isomorphic`(G1, G2)	Returns False if graphs are definitely not isomorphic.
`fast_could_be_isomorphic`(G1, G2)	Returns False if graphs are definitely not isomorphic.
`faster_could_be_isomorphic`(G1, G2)	Returns False if graphs are definitely not isomorphic.

4.15.1 networkx.is_isomorphic

is_isomorphic(*G1, G2, weighted=False, rtol=9.9999999999999995e-07, atol=1.0000000000000001e-09*)

Returns True if the graphs G1 and G2 are isomorphic and False otherwise.

Parameters G1, G2: NetworkX graph instances :

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

weighted: bool, optional :

Optionally check isomorphism for weighted graphs. G1 and G2 must be valid weighted graphs.

rtol: float, optional :

The relative error tolerance when checking weighted edges

atol: float, optional :

The absolute error tolerance when checking weighted edges

See Also:

`isomorphvf2`

Notes

Uses the vf2 algorithm. Works for Graph, DiGraph, MultiGraph, and MultiDiGraph

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

`GraphMatcher.__init__`(G1, G2)	Initialize GraphMatcher.
`GraphMatcher.initialize`()	Reinitializes the state of the algorithm.
`GraphMatcher.is_isomorphic`()	Returns True if G1 and G2 are isomorphic graphs.
`GraphMatcher.subgraph_is_isomorphic`()	Returns True if a subgraph of G1 is isomorphic to G2.
`GraphMatcher.isomorphisms_iter`()	Generator over isomorphisms between G1 and G2.
`GraphMatcher.subgraph_isomorphisms_iter`()	Generator over isomorphisms between a subgraph of G1 and G2.
`GraphMatcher.candidate_pairs_iter`()	Iterator over candidate pairs of nodes in G1 and G2.
`GraphMatcher.match`()	Extends the isomorphism mapping.
`GraphMatcher.semantic_feasibility`(G1_node, ...)	Returns True if adding (G1_node, G2_node) is symantically feasible.
`GraphMatcher.syntactic_feasibility`(G1_node, ...)	Returns True if adding (G1_node, G2_node) is syntactically feasible.

networkx.GraphMatcher.__init__

`__init__`(*G1, G2*)

Initialize GraphMatcher.

Parameters G1,G2: NetworkX Graph or MultiGraph instances. :

The two graphs to check for isomorphism.

Examples

To create a GraphMatcher which checks for syntactic feasibility:

```
>>> G1 = nx.path_graph(4)
>>> G2 = nx.path_graph(4)
>>> GM = nx.GraphMatcher(G1,G2)
```

networkx.GraphMatcher.initialize

`initialize`()

Reinitializes the state of the algorithm.

This method should be redefined if using something other than GMState. If only subclassing GraphMatcher, a redefinition is not necessary.

networkx.GraphMatcher.is_isomorphic

`is_isomorphic`()

Returns True if G1 and G2 are isomorphic graphs.

networkx.GraphMatcher.subgraph_is_isomorphic

`subgraph_is_isomorphic`()

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

networkx.GraphMatcher.isomorphisms_iter

isomorphisms_iter()

Generator over isomorphisms between G1 and G2.

networkx.GraphMatcher.subgraph_isomorphisms_iter

subgraph_isomorphisms_iter()

Generator over isomorphisms between a subgraph of G1 and G2.

networkx.GraphMatcher.candidate_pairs_iter

candidate_pairs_iter()

Iterator over candidate pairs of nodes in G1 and G2.

networkx.GraphMatcher.match

match()

Extends the isomorphism mapping.

This function is called recursively to determine if a complete isomorphism can be found between G1 and G2. It cleans up the class variables after each recursive call. If an isomorphism is found, we yield the mapping.

networkx.GraphMatcher.semantic_feasibility

semantic_feasibility(*G1_node, G2_node*)

Returns True if adding (G1_node, G2_node) is symantically feasible.

The semantic feasibility function should return True if it is acceptable to add the candidate pair (G1_node, G2_node) to the current partial isomorphism mapping. The logic should focus on semantic information contained in the edge data or a formalized node class.

By acceptable, we mean that the subsequent mapping can still become a complete isomorphism mapping. Thus, if adding the candidate pair definitely makes it so that the subsequent mapping cannot become a complete isomorphism mapping, then this function must return False.

The default semantic feasibility function always returns True. The effect is that semantics are not considered in the matching of G1 and G2.

The semantic checks might differ based on the what type of test is being performed. A keyword description of the test is stored in self.test. Here is a quick description of the currently implemented tests:

test='graph' Indicates that the graph matcher is looking for a graph-graph isomorphism.

test='subgraph' Indicates that the graph matcher is looking for a subgraph-graph isomorphism such that a subgraph of G1 is isomorphic to G2.

Any subclass which redefines semantic_feasibility() must maintain the above form to keep the match() method functional. Implementations should consider multigraphs.

networkx.GraphMatcher.syntactic_feasibility

syntactic_feasibility(*G1_node, G2_node*)

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

This function returns True if it is adding the candidate pair to the current partial isomorphism mapping is allowable. The addition is allowable if the inclusion of the candidate pair does not make it impossible for an isomorphism to be found.

DiGraph Matcher

`DiGraphMatcher.__init__`(G1, G2)	Initialize DiGraphMatcher.
`DiGraphMatcher.initialize`()	Reinitializes the state of the algorithm.
`DiGraphMatcher.is_isomorphic`()	Returns True if G1 and G2 are isomorphic graphs.
`DiGraphMatcher.subgraph_is_isomorphic`()	Returns True if a subgraph of G1 is isomorphic to G2.
`DiGraphMatcher.isomorphisms_iter`()	Generator over isomorphisms between G1 and G2.
`DiGraphMatcher.subgraph_isomorphisms_iter`()	Generator over isomorphisms between a subgraph of G1 and G2.
`DiGraphMatcher.candidate_pairs_iter`()	Iterator over candidate pairs of nodes in G1 and G2.
`DiGraphMatcher.match`()	Extends the isomorphism mapping.
`DiGraphMatcher.semantic_feasibility`(G1_node, ...)	Returns True if adding (G1_node, G2_node) is symantically feasible.
`DiGraphMatcher.syntactic_feasibility`(...)	Returns True if adding (G1_node, G2_node) is syntactically feasible.

networkx.DiGraphMatcher.__init__

`__init__`(*G1, G2*)

Initialize DiGraphMatcher.

G1 and G2 should be nx.Graph or nx.MultiGraph instances.

Examples

To create a GraphMatcher which checks for syntactic feasibility:

```
>>> G1 = nx.DiGraph(nx.path_graph(4, create_using=nx.DiGraph()))
>>> G2 = nx.DiGraph(nx.path_graph(4, create_using=nx.DiGraph()))
>>> DiGM = nx.DiGraphMatcher(G1,G2)
```

networkx.DiGraphMatcher.initialize

`initialize`()

Reinitializes the state of the algorithm.

This method should be redefined if using something other than DiGMState. If only subclassing GraphMatcher, a redefinition is not necessary.

networkx.DiGraphMatcher.is_isomorphic

`is_isomorphic`()

Returns True if G1 and G2 are isomorphic graphs.

networkx.DiGraphMatcher.subgraph_is_isomorphic

`subgraph_is_isomorphic`()

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

networkx.DiGraphMatcher.isomorphisms_iter

`isomorphisms_iter`()

Generator over isomorphisms between G1 and G2.

networkx.DiGraphMatcher.subgraph_isomorphisms_iter

subgraph_isomorphisms_iter()

Generator over isomorphisms between a subgraph of G1 and G2.

networkx.DiGraphMatcher.candidate_pairs_iter

candidate_pairs_iter()

Iterator over candidate pairs of nodes in G1 and G2.

networkx.DiGraphMatcher.match

match()

Extends the isomorphism mapping.

This function is called recursively to determine if a complete isomorphism can be found between G1 and G2. It cleans up the class variables after each recursive call. If an isomorphism is found, we yield the mapping.

networkx.DiGraphMatcher.semantic_feasibility

semantic_feasibility(*G1_node, G2_node*)

Returns True if adding (G1_node, G2_node) is symantically feasible.

The semantic feasibility function should return True if it is acceptable to add the candidate pair (G1_node, G2_node) to the current partial isomorphism mapping. The logic should focus on semantic information contained in the edge data or a formalized node class.

By acceptable, we mean that the subsequent mapping can still become a complete isomorphism mapping. Thus, if adding the candidate pair definitely makes it so that the subsequent mapping cannot become a complete isomorphism mapping, then this function must return False.

The default semantic feasibility function always returns True. The effect is that semantics are not considered in the matching of G1 and G2.

The semantic checks might differ based on the what type of test is being performed. A keyword description of the test is stored in self.test. Here is a quick description of the currently implemented tests:

test='graph' Indicates that the graph matcher is looking for a graph-graph isomorphism.

test='subgraph' Indicates that the graph matcher is looking for a subgraph-graph isomorphism such that a subgraph of G1 is isomorphic to G2.

Any subclass which redefines semantic_feasibility() must maintain the above form to keep the match() method functional. Implementations should consider multigraphs.

networkx.DiGraphMatcher.syntactic_feasibility

syntactic_feasibility(*G1_node, G2_node*)

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

This function returns True if it is adding the candidate pair to the current partial isomorphism mapping is allowable. The addition is allowable if the inclusion of the candidate pair does not make it impossible for an isomorphism to be found.

Weighted Graph Matcher

WeightedGraphMatcher.__init__(G1, G2[, ...])	Initialize WeightedGraphMatcher.
WeightedGraphMatcher.initialize()	Reinitializes the state of the algorithm.
WeightedGraphMatcher.is_isomorphic()	Returns True if G1 and G2 are isomorphic graphs.
WeightedGraphMatcher.subgraph_is_isomorphic()	Returns True if a subgraph of G1 is isomorphic to G2.
WeightedGraphMatcher.isomorphisms_iter()	Generator over isomorphisms between G1 and G2.
WeightedGraphMatcher.subgraph_isomorphisms_iter()	Generator over isomorphisms between a subgraph of G1 and G2.
WeightedGraphMatcher.candidate_pairs_iter()	Iterator over candidate pairs of nodes in G1 and G2.
WeightedGraphMatcher.match()	Extends the isomorphism mapping.
WeightedGraphMatcher.semantic_feasibility(...)	Returns True if mapping G1_node to G2_node is semantically feasible.
WeightedGraphMatcher.syntactic_feasibility(...)	Returns True if adding (G1_node, G2_node) is syntactically feasible.

networkx.WeightedGraphMatcher.__init__

__init__(*G1, G2, rtol=9.9999999999999995e-07, atol=1.0000000000000001e-09*)

Initialize WeightedGraphMatcher.

Parameters **G1, G2** : nx.Graph instances

G1 and G2 must be weighted graphs.

rtol : float, optional

The relative tolerance used to compare weights.

atol : float, optional

The absolute tolerance used to compare weights.

networkx.WeightedGraphMatcher.initialize

initialize()

Reinitializes the state of the algorithm.

This method should be redefined if using something other than GMState. If only subclassing GraphMatcher, a redefinition is not necessary.

networkx.WeightedGraphMatcher.is_isomorphic

is_isomorphic()

Returns True if G1 and G2 are isomorphic graphs.

networkx.WeightedGraphMatcher.subgraph_is_isomorphic

subgraph_is_isomorphic()

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

networkx.WeightedGraphMatcher.isomorphisms_iter

isomorphisms_iter()

Generator over isomorphisms between G1 and G2.

networkx.WeightedGraphMatcher.subgraph_isomorphisms_iter

`subgraph_isomorphisms_iter`()

Generator over isomorphisms between a subgraph of G1 and G2.

networkx.WeightedGraphMatcher.candidate_pairs_iter

`candidate_pairs_iter`()

Iterator over candidate pairs of nodes in G1 and G2.

networkx.WeightedGraphMatcher.match

`match`()

Extends the isomorphism mapping.

This function is called recursively to determine if a complete isomorphism can be found between G1 and G2. It cleans up the class variables after each recursive call. If an isomorphism is found, we yield the mapping.

networkx.WeightedGraphMatcher.semantic_feasibility

`semantic_feasibility`(*G1_node, G2_node*)

Returns True if mapping G1_node to G2_node is semantically feasible.

networkx.WeightedGraphMatcher.syntactic_feasibility

`syntactic_feasibility`(*G1_node, G2_node*)

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

This function returns True if it is adding the candidate pair to the current partial isomorphism mapping is allowable. The addition is allowable if the inclusion of the candidate pair does not make it impossible for an isomorphism to be found.

Weighted DiGraph Matcher

`WeightedDiGraphMatcher.__init__`(G1, G2[, ...])	Initialize WeightedGraphMatcher.
`WeightedDiGraphMatcher.initialize`()	Reinitializes the state of the algorithm.
`WeightedDiGraphMatcher.is_isomorphic`()	Returns True if G1 and G2 are isomorphic graphs.
`WeightedDiGraphMatcher.subgraph_is_isomorphic`()	Returns True if a subgraph of G1 is isomorphic to G2.
`WeightedDiGraphMatcher.isomorphisms_iter`()	Generator over isomorphisms between G1 and G2.
`WeightedDiGraphMatcher.subgraph_isomorphisms_iter`()	Generator over isomorphisms between a subgraph of G1 and G2.
`WeightedDiGraphMatcher.candidate_pairs_iter`()	Iterator over candidate pairs of nodes in G1 and G2.
`WeightedDiGraphMatcher.match`()	Extends the isomorphism mapping.
`WeightedDiGraphMatcher.semantic_feasibility`(...)	Returns True if mapping G1_node to G2_node is semantically feasible.
`WeightedDiGraphMatcher.syntactic_feasibility`(...)	Returns True if adding (G1_node, G2_node) is syntactically feasible.

networkx.WeightedDiGraphMatcher.__init__

`__init__`(*G1, G2, rtol=9.9999999999999995e-07, atol=1.0000000000000001e-09*)

Initialize WeightedGraphMatcher.

Parameters **G1, G2** : nx.DiGraph instances

G1 and G2 must be weighted graphs.

rtol : float, optional

The relative tolerance used to compare weights.

atol : float, optional

The absolute tolerance used to compare weights.

networkx.WeightedDiGraphMatcher.initialize

initialize()

Reinitializes the state of the algorithm.

This method should be redefined if using something other than DiGMState. If only subclassing GraphMatcher, a redefinition is not necessary.

networkx.WeightedDiGraphMatcher.is_isomorphic

is_isomorphic()

Returns True if G1 and G2 are isomorphic graphs.

networkx.WeightedDiGraphMatcher.subgraph_is_isomorphic

subgraph_is_isomorphic()

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

networkx.WeightedDiGraphMatcher.isomorphisms_iter

isomorphisms_iter()

Generator over isomorphisms between G1 and G2.

networkx.WeightedDiGraphMatcher.subgraph_isomorphisms_iter

subgraph_isomorphisms_iter()

Generator over isomorphisms between a subgraph of G1 and G2.

networkx.WeightedDiGraphMatcher.candidate_pairs_iter

candidate_pairs_iter()

Iterator over candidate pairs of nodes in G1 and G2.

networkx.WeightedDiGraphMatcher.match

match()

Extends the isomorphism mapping.

This function is called recursively to determine if a complete isomorphism can be found between G1 and G2. It cleans up the class variables after each recursive call. If an isomorphism is found, we yield the mapping.

networkx.WeightedDiGraphMatcher.semantic_feasibility

semantic_feasibility(*G1_node, G2_node*)

Returns True if mapping G1_node to G2_node is semantically feasible.

networkx.WeightedDiGraphMatcher.syntactic_feasibility

syntactic_feasibility(*G1_node, G2_node*)

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

This function returns True if it is adding the candidate pair to the current partial isomorphism mapping is allowable. The addition is allowable if the inclusion of the candidate pair does not make it impossible for an isomorphism to be found.

Weighted MultiGraph Matcher

WeightedMultiGraphMatcher.__init__(G1, G2[, ...])	Initialize WeightedGraphMatcher.
WeightedMultiGraphMatcher.initialize()	Reinitializes the state of the algorithm.
WeightedMultiGraphMatcher.is_isomorphic()	Returns True if G1 and G2 are isomorphic graphs.
WeightedMultiGraphMatcher.subgraph_is_isomorphic()	Returns True if a subgraph of G1 is isomorphic to G2.
WeightedMultiGraphMatcher.isomorphisms_iter()	Generator over isomorphisms between G1 and G2.
WeightedMultiGraphMatcher.subgraph_isomorphisms_iter()	Generator over isomorphisms between a subgraph of G1 and G2.
WeightedMultiGraphMatcher.candidate_pairs_iter()	Iterator over candidate pairs of nodes in G1 and G2.
WeightedMultiGraphMatcher.match()	Extends the isomorphism mapping.
WeightedMultiGraphMatcher.semantic_feasibility(...)	Returns True if mapping G1_node to G2_node is semantically feasible.
WeightedMultiGraphMatcher.syntactic_feasibility(...)	Returns True if adding (G1_node, G2_node) is syntactically feasible.

networkx.WeightedMultiGraphMatcher.__init__

__init__(*G1, G2, rtol=9.9999999999999995e-07, atol=1.0000000000000001e-09*)

Initialize WeightedGraphMatcher.

Parameters **G1, G2** : nx.MultiGraph instances

G1 and G2 must be weighted graphs.

rtol : float, optional

The relative tolerance used to compare weights.

atol : float, optional

The absolute tolerance used to compare weights.

networkx.WeightedMultiGraphMatcher.initialize

initialize()

Reinitializes the state of the algorithm.

This method should be redefined if using something other than GMState. If only subclassing GraphMatcher, a redefinition is not necessary.

networkx.WeightedMultiGraphMatcher.is_isomorphic

is_isomorphic()

Returns True if G1 and G2 are isomorphic graphs.

networkx.WeightedMultiGraphMatcher.subgraph_is_isomorphic

subgraph_is_isomorphic()

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

networkx.WeightedMultiGraphMatcher.isomorphisms_iter

isomorphisms_iter()

Generator over isomorphisms between G1 and G2.

networkx.WeightedMultiGraphMatcher.subgraph_isomorphisms_iter

subgraph_isomorphisms_iter()

Generator over isomorphisms between a subgraph of G1 and G2.

networkx.WeightedMultiGraphMatcher.candidate_pairs_iter

candidate_pairs_iter()

Iterator over candidate pairs of nodes in G1 and G2.

networkx.WeightedMultiGraphMatcher.match

match()

Extends the isomorphism mapping.

This function is called recursively to determine if a complete isomorphism can be found between G1 and G2. It cleans up the class variables after each recursive call. If an isomorphism is found, we yield the mapping.

networkx.WeightedMultiGraphMatcher.semantic_feasibility

semantic_feasibility(*G1_node, G2_node*)

Returns True if mapping G1_node to G2_node is semantically feasible.

networkx.WeightedMultiGraphMatcher.syntactic_feasibility

syntactic_feasibility(*G1_node, G2_node*)

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

This function returns True if it is adding the candidate pair to the current partial isomorphism mapping is allowable. The addition is allowable if the inclusion of the candidate pair does not make it impossible for an isomorphism to be found.

Weighted MultiDiGraph Matcher

WeightedMultiDiGraphMatcher.__init__(G1, G2)	Initialize WeightedGraphMatcher.
WeightedMultiDiGraphMatcher.initialize()	Reinitializes the state of the algorithm.
WeightedMultiDiGraphMatcher.is_isomorphic()	Returns True if G1 and G2 are isomorphic graphs.
WeightedMultiDiGraphMatcher.subgraph_is_isomorphic()	Returns True if a subgraph of G1 is isomorphic to G2.
WeightedMultiDiGraphMatcher.isomorphisms_iter()	Generator over isomorphisms between G1 and G2.
WeightedMultiDiGraphMatcher.subgraph_isomorphisms_iter()	Generator over isomorphisms between a subgraph of G1 and G2.
WeightedMultiDiGraphMatcher.candidate_pairs_iter()	Iterator over candidate pairs of nodes in G1 and G2.
WeightedMultiDiGraphMatcher.match()	Extends the isomorphism mapping.
WeightedMultiDiGraphMatcher.semantic_feasibility(...)	Returns True if mapping G1_node to G2_node is semantically feasible.
WeightedMultiDiGraphMatcher.syntactic_feasibility(...)	Returns True if adding (G1_node, G2_node) is syntactically feasible.

networkx.WeightedMultiDiGraphMatcher.__init__

`__init__`(*G1, G2, rtol=9.9999999999999995e-07, atol=1.0000000000000001e-09*)

Initialize WeightedGraphMatcher.

Parameters G1, G2 : nx.MultiDiGraph instances

G1 and G2 must be weighted graphs.

rtol : float, optional

The relative tolerance used to compare weights.

atol : float, optional

The absolute tolerance used to compare weights.

networkx.WeightedMultiDiGraphMatcher.initialize

`initialize`()

Reinitializes the state of the algorithm.

This method should be redefined if using something other than DiGMState. If only subclassing GraphMatcher, a redefinition is not necessary.

networkx.WeightedMultiDiGraphMatcher.is_isomorphic

`is_isomorphic`()

Returns True if G1 and G2 are isomorphic graphs.

networkx.WeightedMultiDiGraphMatcher.subgraph_is_isomorphic

`subgraph_is_isomorphic`()

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

networkx.WeightedMultiDiGraphMatcher.isomorphisms_iter

`isomorphisms_iter`()

Generator over isomorphisms between G1 and G2.

networkx.WeightedMultiDiGraphMatcher.subgraph_isomorphisms_iter

subgraph_isomorphisms_iter()

Generator over isomorphisms between a subgraph of G1 and G2.

networkx.WeightedMultiDiGraphMatcher.candidate_pairs_iter

candidate_pairs_iter()

Iterator over candidate pairs of nodes in G1 and G2.

networkx.WeightedMultiDiGraphMatcher.match

match()

Extends the isomorphism mapping.

This function is called recursively to determine if a complete isomorphism can be found between G1 and G2. It cleans up the class variables after each recursive call. If an isomorphism is found, we yield the mapping.

networkx.WeightedMultiDiGraphMatcher.semantic_feasibility

semantic_feasibility(*G1_node, G2_node*)

Returns True if mapping G1_node to G2_node is semantically feasible.

networkx.WeightedMultiDiGraphMatcher.syntactic_feasibility

syntactic_feasibility(*G1_node, G2_node*)

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

This function returns True if it is adding the candidate pair to the current partial isomorphism mapping is allowable. The addition is allowable if the inclusion of the candidate pair does not make it impossible for an isomorphism to be found.

4.16 Link Analysis

4.16.1 PageRank

PageRank analysis of graph structure.

pagerank(G[, alpha, max_iter, tol, nstart])	Return the PageRank of the nodes in the graph.
pagerank_numpy(G[, alpha])	Return the PageRank of the nodes in the graph.
pagerank_scipy(G[, alpha, max_iter, tol, ...])	Return the PageRank of the nodes in the graph.
google_matrix(G[, alpha, nodelist])	Return the Google matrix of the graph.

networkx.pagerank

pagerank(*G, alpha=0.84999999999999998, max_iter=100, tol=1e-08, nstart=None*)

Return the PageRank of the nodes in the graph.

PageRank computes 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

[R95], [R96]

Examples

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

networkx.pagerank_numpy

pagerank_numpy(*G, alpha=0.84999999999999998*)

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

[R97], [R98]

Examples

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

networkx.pagerank_scipy

pagerank_scipy(*G, alpha=0.84999999999999998, max_iter=100, tol=9.9999999999999995e-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

[R99], [R100]

Examples

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

networkx.google_matrix

google_matrix(*G, alpha=0.84999999999999998, 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.

hits(G[, max_iter, tol, nstart])	Return HITS hubs and authorities values for nodes.
hits_numpy(G)	Return HITS hubs and authorities values for nodes.
hits_scipy(G[, max_iter, tol])	Return HITS hubs and authorities values for nodes.
hub_matrix(G[, nodelist])	Return the HITS hub matrix.
authority_matrix(G[, nodelist])	Return the HITS authority matrix.

networkx.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

[R86], [R87]

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

[R88], [R89]

Examples

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

networkx.hits_scipy

hits_scipy (*G, max_iter=100, tol=9.9999999999999995e-07*)
Return HITS hubs and authorities values for nodes.

The HITS algorithm computes two numbers for a node. Authorities estimates the node value based on the incoming links. Hubs estimates the node value based on outgoing links.

Parameters G : graph

A NetworkX graph

max_iter : interger, optional

Maximum number of iterations in power method.

tol : float, optional

Error tolerance used to check convergence in power method iteration.

nstart : dictionary, optional

Starting value of each node for power method iteration.

Returns (hubs,authorities) : two-tuple of dictionaries

Two dictionaries keyed by node containing the hub and authority values.

Notes

This implementation uses SciPy sparse matrices.

The eigenvector calculation is done by the power iteration method and has no guarantee of convergence. The iteration will stop after max_iter iterations or an error tolerance of number_of_nodes(G)*tol has been reached.

The HITS algorithm was designed for directed graphs but this algorithm does not check if the input graph is directed and will execute on undirected graphs.

References

[R90], [R91]

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(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

[R93]

4.18 Mixing Patterns

Mixing matrices and assortativity coefficients.

4.18.1 Assortativity

degree_assortativity(G)	Compute degree assortativity of graph.
attribute_assortativity(G, attribute)	Compute assortativity for node attributes.
numeric_assortativity(G, attribute)	Compute assortativity for numerical node attributes.
neighbor_connectivity(G)	Compute neighbor connectivity of graph.
degree_pearsonr(G)	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. [R52] , where e is the joint probability distribution (mixing matrix) of the degrees. If G is directed than the matrix e is the joint probability of out-degree and in-degree.

References

[R52]

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. [R48] , (trace(e)-sum(e))/(1-sum(e)), where e is the joint probability distribution (mixing matrix) of the specified attribute.

References

[R48]

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. [R94] , where e is the joint probability distribution (mixing matrix) of the specified attribute.

References

[R94]

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

[R53]

Examples

```
>>> G=nx.path_graph(4)
>>> r=nx.degree_pearsonr(G)
>>> r
-0.5
```

4.18.2 Mixing

attribute_mixing_matrix(G, attribute[, ...])	Return mixing matrix for attribute.
degree_mixing_matrix(G[, normalized])	Return mixing matrix for attribute.
degree_mixing_dict(G[, normalized])	Return dictionary representation of mixing matrix for degree.
attribute_mixing_dict(G, attribute[, normalized])	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)	Return a minimum spanning tree or forest of an undirected weighted graph.
minimum_spanning_edges(G)	Generate edges in a minimum spanning forest of an undirected weighted graph.

4.19.1 networkx.minimum_spanning_tree

minimum_spanning_tree(*G*)

Return a minimum spanning tree or forest of an undirected weighted graph.

A minimum spanning tree is a subgraph of the graph (a tree) with the minimum sum of edge weights.

If the graph is not connected a spanning forest is constructed. A spanning forest is a union of the spanning trees for each connected component of the graph.

Parameters G : NetworkX Graph

Returns G : NetworkX Graph

A minimum spanning tree or forest.

Notes

Uses Kruskal's algorithm.

If the graph edges do not have a weight attribute a default weight of 1 will be assigned.

Examples

```
>>> G=nx.cycle_graph(4)
>>> G.add_edge(0,3,weight=2) # assign weight 2 to edge 0-3
>>> T=nx.minimum_spanning_tree(G)
>>> print(sorted(T.edges(data=True)))
[(0, 1, {'weight': 1}), (1, 2, {'weight': 1}), (2, 3, {'weight': 1})]
```

4.19.2 networkx.minimum_spanning_edges

minimum_spanning_edges(*G*)

Generate edges in a minimum spanning forest of an undirected weighted graph.

A minimum spanning tree is a subgraph of the graph (a tree) with the minimum sum of edge weights. A spanning forest is a union of the spanning trees for each connected component of the graph.

Parameters G : NetworkX Graph

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_edges(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.

cartesian_product(G, H[, create_using])	Return the Cartesian product of G and H.
compose(G, H[, create_using, name])	Return a new graph of G composed with H.
complement(G[, create_using, name])	Return graph complement of G.
union(G, H[, create_using, rename, name])	Return the union of graphs G and H.
disjoint_union(G, H)	Return the disjoint union of graphs G and H, forcing distinct integer node labels.
intersection(G, H[, create_using])	Return a new graph that contains only the edges that exist in both G and H.
difference(G, H[, create_using])	Return a new graph that contains the edges that exist in G but not in H.
symmetric_difference(G, H[, create_using])	Return new graph with edges that exist in either G or H but not both.

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 be specifying the tuple rename=('G-','H-') (for example). Node u in G is then renamed "G-u" and v in H is renamed "H-v".

name : string

Specify the name for the union graph

See Also:

`disjoint_union`

Notes

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

Graph, edge, and node attributes are propagated from G and H to the union graph. If a graph attribute is present in both G and H the value from G is used.

4.20.5 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 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).

`shortest_path`(G[, source, target, weighted])	Compute shortest paths in the graph.
`shortest_path_length`(G[, source, target, ...])	Compute shortest path lengths in the graph.
`average_shortest_path_length`(G[, weighted])	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 pats lenghts for all connected node pairs.

target : node, optional

Ending node for path. If not specified compute shortest path lenghts 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.

single_source_shortest_path(G, source[, cutoff])	Compute shortest path between source and all other nodes reachable from source.
single_source_shortest_path_length(G, source)	Compute the shortest path lengths from source to all reachable nodes.
all_pairs_shortest_path(G[, cutoff])	Compute shortest paths between all nodes.
all_pairs_shortest_path_length(G[, cutoff])	Compute the shortest path lengths between all nodes in G.
predecessor(G, source[, target, cutoff, ...])	Returns dictionary of predecessors for the path from source to all nodes in G.
floyd_warshall(G)	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 <= 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 <= 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 <= 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 <= 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 <= 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.

dijkstra_path(G, source, target[, weight])	Returns the shortest path from source to target in a weighted graph G.
dijkstra_path_length(G, source, target[, weight])	Returns the shortest path length from source to target in a weighted graph G.
single_source_dijkstra_path(G, source[, weight])	Compute shortest path between source and all other reachable nodes for a weighted graph.
single_source_dijkstra_path_length(G, source)	Compute shortest path length between source and all other reachable nodes for a weighted graph.
all_pairs_dijkstra_path(G[, weight])	Compute shortest paths between all nodes in a weighted graph.
all_pairs_dijkstra_path_length(G[, weight])	Compute shortest path lengths between all nodes in a weighted graph.
single_source_dijkstra(G, source[, target, ...])	Compute shortest paths and lengths in a weighted graph G.
bidirectional_dijkstra(G, source, target[, ...])	Dijkstra's algorithm for shortest paths using bidirectional search.
bidirectional_shortest_path(G, source, target)	Return a list of nodes in a shortest path between source and target.
dijkstra_predecessor_and_distance(G, source)	Compute shorest path length and predecessors on shortest paths in weighted graphs.
bellman_ford(G, source[, weight])	Compute shortest path lengths and predecessors on shortest paths in weighted graphs.

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 <= 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 dicdtionary 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 shortest path length and predecessors on shortest paths in weighted graphs.

Parameters **G** : NetworkX graph

source : node label

Starting node for path

weight: string, optional :

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.

networkx.bellman_ford

bellman_ford(*G, source, weight='weight'*)

Compute shortest path lengths and predecessors on shortest paths in weighted graphs.

The algorithm has a running time of O(mn) where n is the number of nodes and n is the number of edges.

Parameters **G** : NetworkX graph

The algorithm works for all types of graphs, including directed graphs and multigraphs.

source: node label :

Starting node for path

weight: string, optional :

Edge data key corresponding to the edge weight

Returns **pred,dist** : dictionaries

Returns two dictionaries representing a list of predecessors of a node and the distance from the source to each node. The dictionaries are keyed by target node label.

Raises **NetworkXError** :

If the (di)graph contains a negative cost (di)cycle, the algorithm raises an exception to indicate the presence of the negative cost (di)cycle.

Notes

The dictionaries returned only have keys for nodes reachable from the source.

In the case where the (di)graph is not connected, if a component not containing the source contains a negative cost (di)cycle, it will not be detected.

Examples

```
>>> import networkx as nx
>>> G = nx.path_graph(5, create_using = nx.DiGraph())
>>> pred, dist = nx.bellman_ford(G, 0)
>>> pred
{0: None, 1: 0, 2: 1, 3: 2, 4: 3}
>>> dist
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4}
```

```
>>> from nose.tools import assert_raises
>>> G = nx.cycle_graph(5)
>>> G[1][2]['weight'] = -7
>>> assert_raises(nx.NetworkXError, nx.bellman_ford, G, 0)
```

4.21.5 A* Algorithm

Shortest paths and path lengths using A* (“A star”) algorithm.

astar_path(G, source, target[, heuristic])	Return a list of nodes in a shortest path between source and target using the A* (“A-star”) algorithm.
astar_path_length(G, source, target[, heuristic])	Return a list of nodes in a shortest path between source and target using the A* (“A-star”) algorithm.

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(a, b):
...    (x1, y1) = a
...    (x2, y2) = b
...    return ((x1 - x2) ** 2 + (y1 - y2) ** 2) ** 0.5
>>> print(nx.astar_path(G,(0,0),(2,2),dist))
[(0, 0), (0, 1), (1, 1), (1, 2), (2, 2)]
```

networkx.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.

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

networkx.dfs_preorder

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

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

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

networkx.dfs_postorder

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

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

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

networkx.dfs_predecessor

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

Return predecessors of depth-first-search with root at source.

networkx.dfs_successor

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

Return succesors of depth-first-search with root at source.

networkx.dfs_tree

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

Return directed graph (tree) of depth-first-search with root at source.

If the graph is disconnected, return a disconnected graph (forest).

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}
```

CHAPTER

FIVE

FUNCTIONS

Functional interface to graph methods and assorted utilities.

5.1 Graph functions

`density`(G)	Return the density of a graph.
`info`(G[, n])	Print short summary of information for the graph G or the node n.
`degree_histogram`(G)	Return a list of the frequency of each degree value.
`freeze`(G)	Modify graph to prevent addition of nodes or edges.
`is_frozen`(G)	Return True if graph is frozen.
`create_empty_copy`(G[, with_nodes])	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*)

Print short summary of information for the graph G or the node n.

Parameters **G** : Networkx graph

A graph

n : node (any hashable)

A node in the graph G

5.1.3 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.Graph()
>>> G.add_path([0,1,2,3])
>>> G=nx.freeze(G)
>>> try:
...     G.add_edge(4,5)
... except nx.NetworkXError as e:
...     print(str(e))
Frozen graph can't be modified
```

5.1.5 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.

CHAPTER

SIX

GRAPH GENERATORS

6.1 Atlas

Generators for the small graph atlas.

See "An Atlas of Graphs" by Ronald C. Read and Robin J. Wilson, Oxford University Press, 1998.

Because of its size, this module is not imported by default.

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

6.1.1 networkx.generators.atlas.graph_atlas_g

graph_atlas_g()

Return the list [G0,G1,...,G1252] of graphs as named in the Graph Atlas. G0,G1,...,G1252 are all graphs with up to 7 nodes.

The graphs are listed:

1. in increasing order of number of nodes;
2. for a fixed number of nodes, in increasing order of the number of edges;
3. for fixed numbers of nodes and edges, in increasing order of the degree sequence, for example 111223 < 112222;
4. for fixed degree sequence, in increasing number of automorphisms.

Note that indexing is set up so that for GAG=graph_atlas_g(), then G123=GAG[123] and G[0]=empty_graph(0)

6.2 Classic

Generators for some classic graphs.

The typical graph generator is called as follows:

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

returning the complete graph on n nodes labeled 0,..,99 as a simple graph. Except for empty_graph, all the generators in this module return a Graph class (i.e. a simple, undirected graph).

balanced_tree(r, h[, create_using])	Return the perfectly balanced r-tree of height h.
barbell_graph(m1, m2[, create_using])	Return the Barbell Graph: two complete graphs connected by a path.
complete_graph(n[, create_using])	Return the Complete graph K_n with n nodes.
complete_bipartite_graph(n1, n2[, create_using])	Return the complete bipartite graph K_{n1_n2}.
circular_ladder_graph(n[, create_using])	Return the circular ladder graph CL_n of length n.
cycle_graph(n[, create_using])	Return the cycle graph C_n over n nodes.
dorogovtsev_goltsev_mendes_graph(n[, ...])	Return the hierarchically constructed Dorogovtsev-Goltsev-Mendes graph.
empty_graph([n, create_using])	Return the empty graph with n nodes and zero edges.
grid_2d_graph(m, n[, periodic, create_using])	Return the 2d grid graph of mxn nodes, each connected to its nearest neighbors.
grid_graph(dim[, periodic, create_using])	Return the n-dimensional grid graph.
hypercube_graph(n[, create_using])	Return the n-dimensional hypercube.
ladder_graph(n[, create_using])	Return the Ladder graph of length n.
lollipop_graph(m, n[, create_using])	Return the Lollipop Graph; K_m connected to P_n.
null_graph([create_using])	Return the Null graph with no nodes or edges.
path_graph(n[, create_using])	Return the Path graph P_n of n nodes linearly connected by n-1 edges.
star_graph(n[, create_using])	Return the Star graph with n+1 nodes: one center node, connected to n outer nodes.
trivial_graph([create_using])	Return the Trivial graph with one node (with integer label 0) and no edges.
wheel_graph(n[, create_using])	Return the wheel graph: a single hub node connected to each node of the (n-1)-node cycle graph.

6.2.1 networkx.generators.classic.balanced_tree

balanced_tree (*r, h, create_using=None*)

Return the perfectly balanced r-tree of height h.

For r>=2, h>=1, this is the rooted tree where all leaves are at distance h from the root. The root has degree r and all other internal nodes have degree r+1.

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

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

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 >= 0.

Two identical complete graphs K_{m1} form the left and right bells, and are connected by a path P_{m2}.

The 2*m1+m2 nodes are numbered 0,...,m1-1 for the left barbell, m1,...,m1+m2-1 for the path, and m1+m2,...,2*m1+m2-1 for the right barbell.

The 3 subgraphs are joined via the edges (m1-1,m1) and (m1+m2-1,m1+m2). If m2=0, this is merely two complete graphs joined together.

This graph is an extremal example in David Aldous and Jim Fill's etext on Random Walks on Graphs.

6.2.3 networkx.generators.classic.complete_graph

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 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 mxn nodes, each connected to its nearest neighbors. Optional argument periodic=True will connect boundary nodes via periodic boundary conditions.

6.2.10 networkx.generators.classic.grid_graph

grid_graph (*dim, periodic=False, create_using=None*)

Return the n-dimensional grid graph.

The dimension is the length of the list 'dim' and the size in each dimension is the value of the list element.

E.g. G=grid_graph(dim=[2,3]) produces a 2x3 grid graph.

If periodic=True then join grid edges with periodic boundary conditions.

6.2.11 networkx.generators.classic.hypercube_graph

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>=0, the complete graph K_m is connected to the path P_n. The resulting m+n nodes are labelled 0,...,m-1 for the complete graph and m,...,m+n-1 for the path. The 2 subgraphs are joined via the edge (m-1,m). If n=0, this is merely a complete graph.

Node labels are the integers 0 to number_of_nodes - 1.

(This graph is an extremal example in David Aldous and Jim Fill's etext on Random Walks on Graphs.)

6.2.14 networkx.generators.classic.null_graph

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.

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

6.3.1 networkx.generators.small.make_small_graph

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 Chvátal 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.

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

6.4.1 networkx.generators.random_graphs.fast_gnp_random_graph

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

Return a random graph G_{n,p}.

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

Sometimes called Erdős-Rényi graph, or binomial graph.

Parameters **n** : 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

[R78]

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}.

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

Sometimes called Erdős-Rényi graph, or binomial graph.

Parameters **n** : 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

[R79], [R80]

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

[R74], [R75]

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

[R73]

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}.

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

Sometimes called Erdős-Rényi graph, or binomial graph.

Parameters **n** : 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

[R76], [R77]

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}.

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

Sometimes called Erdős-Rényi graph, or binomial graph.

Parameters **n** : 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

[R71], [R72]

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

[R81]

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

[R85]

6.4.10 networkx.generators.random_graphs.connected_watts_strogatz_graph

connected_watts_strogatz_graph (*n, k, p, tries=100, create_using=None, seed=None*)

Return a connected Watts-Strogatz small-world graph.

Attempt to generate a connected realization by repeated generation of Watts-Strogatz small-world graphs. An exception is raised if the maximum number of tries is exceeded.

Parameters **n** : int

The number of nodes

k : int

Each node is connected to k nearest neighbors in ring topology

p : float

The probability of rewiring each edge

tries : int

Number of attempts to generate a connected graph.

create_using : graph, optional (default Graph)

The graph instance used to build the graph.

seed : int, optional

The seed for random number generator.

See Also:

`newman_watts_strogatz_graph`, `watts_strogatz_graph`

6.4.11 networkx.generators.random_graphs.random_regular_graph

random_regular_graph (*d, n, create_using=None, seed=None*)

Return a random regular graph of n nodes each with degree d.

The resulting graph G has no self-loops or parallel edges.

Parameters **d** : int

Degree

n : integer

Number of nodes. The value of n*d must be even.

create_using : graph, optional (default Graph)

The graph instance used to build the graph.

seed : hashable object

The seed for random number generator.

Notes

The nodes are numbered form 0 to n-1.

Kim and Vu's paper [R84] shows that this algorithm samples in an asymptotically uniform way from the space of random graphs when d = O(n**(1/3-epsilon)).

References

[R83], [R84]

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 with m nodes and no edges.

References

[R70]

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

[R82]

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 or edges in the shell

d : float

The ratio of inter-shell (next) edges to intra-shell edges. d=0 means no intra shell edges, d=1 for the last shell

create_using : graph, optional (default Graph)

The graph instance used to build the graph.

seed : int, optional

Seed for random number generator (default=None).

Examples

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

6.4.16 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.

configuration_model(deg_sequence[, ...])	Return a random graph with the given degree sequence.
directed_configuration_model(...[, ...])	Return a directed_random graph with the given degree sequences.
expected_degree_graph(w[, create_using, seed])	Return a random graph G(w) with expected degrees given by w.
havel_hakimi_graph(deg_sequence[, create_using])	Return a simple graph with given degree sequence, constructed using the Havel-Hakimi algorithm.
degree_sequence_tree(deg_sequence[, ...])	Make a tree for the given degree sequence.
is_valid_degree_sequence(deg_sequence)	Return True if deg_sequence is a valid sequence of integer degrees equal to the degree sequence of some simple graph.
create_degree_sequence(n, **kwds[, ...])	Attempt to create a valid degree sequence of length n using specified function sfunction(n,**kwds).
double_edge_swap(G[, nswap])	Attempt nswap double-edge swaps on the graph G.
connected_double_edge_swap(G[, nswap])	Attempt nswap double-edge swaps on the graph G.
li_smax_graph(degree_seq[, create_using])	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 [R60].

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

[R60]

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 [R62].

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

[R62]

Examples

```
>>> D=nx.DiGraph([(0,1),(1,2),(2,3)]) # directed path graph
>>> din=list(D.in_degree().values())
>>> dout=list(D.out_degree().values())
>>> din.append(1)
>>> dout[0]=2
>>> D=nx.directed_configuration_model(din,dout)
```

To remove parallel edges:

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

To remove self loops:

```
>>> D.remove_edges_from(D.selfloop_edges())
```

6.5.3 networkx.generators.degree_seq.expected_degree_graph

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

[R63]

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

[R64]

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 [R65]. This algorithm is also used in havel_hakimi_graph()

References

[R65]

6.5.7 networkx.generators.degree_seq.create_degree_sequence

create_degree_sequence(*n, sfunction=None, max_tries=50, **kwds*)

Attempt to create a valid degree sequence of length n using specified function sfunction(n,**kwds).

Parameters **n** : int

Length of degree sequence = number of nodes

sfunction: function :

Function which returns a list of n real or integer values. Called as "sfunction(n,**kwds)".

max_tries: int :

Max number of attempts at creating valid degree sequence.

Notes

Repeatedly create a degree sequence by calling sfunction(n,**kwds) until achieving a valid degree sequence. If unsuccessful after max_tries attempts, raise an exception.

For examples of sfunctions that return sequences of random numbers, see networkx.Utils.

Examples

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

6.5.8 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 choseen edges u-v and x-y and creates the new edges u-x and v-y:

```
u--v          u   v
       becomes  |   |
x--y          x   y
```

If either the edge u-x or v-y already exist no swap is performed so the actual count of swapped edges is always <= nswap

Does not enforce any connectivity constraints.

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 choseen edges u-v and x-y and creates the new edges u-x and v-y:

```
u--v          u   v
       becomes  |   |
x--y          x   y
```

If either the edge u-x or v-y already exist no swap is performed so the actual count of swapped edges is always <= nswap

The initial graph G must be connected and the resulting graph is connected.

References

[R61]

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-metrix means that high degree nodes are connected to high degree nodes.

- ***degree_seq*: degree sequence, a list of integers with each entry** corresponding to the degree of a node. A non-graphical degree sequence raises an Exception.

Reference:

```
@unpublished{li-2005,
 author = {Lun Li and David Alderson and Reiko Tanaka
          and John C. Doyle and Walter Willinger},
 title = {Towards a Theory of Scale-Free Graphs:
         Definition, Properties, and  Implications (Extended Version)},
 url = {http://arxiv.org/abs/cond-mat/0501169},
```

```
 year = {2005}
}
```

The algorithm:

```
STEP 0 - Initialization
A = {0}
B = {1, 2, 3, ..., n}
O = {(i; j), ..., (k, l),...} where i < j, i <= k < l and
        d_i * d_j >= d_k *d_l
wA = d_1
dB = sum(degrees)

STEP 1 - Link selection
(a) If |O| = 0 TERMINATE. Return graph A.
(b) Select element(s) (i, j) in O having the largest d_i * d_j , if for
        any i or j either w_i = 0 or w_j = 0 delete (i, j) from O
(c) If there are no elements selected go to (a).
(d) Select the link (i, j) having the largest value w_i (where for each
        (i, j) w_i is the smaller of w_i and w_j ), and proceed to STEP 2.

STEP 2 - Link addition
Type 1: i in A and j in B.
        Add j to the graph A and remove it from the set B add a link
        (i, j) to the graph A. Update variables:
        wA = wA + d_j -2 and dB = dB - d_j
        Decrement w_i and w_j with one. Delete (i, j) from O
Type 2: i and j in A.
    Check Tree Condition: If dB = 2 * |B| - wA.
        Delete (i, j) from O, continue to STEP 3
    Check Disconnected Cluster Condition: If wA = 2.
        Delete (i, j) from O, continue to STEP 3
    Add the link (i, j) to the graph A
    Decrement w_i and w_j with one, and wA = wA -2
STEP 3
    Go to STEP 1
```

The article states that the algorithm will result in a maximal s-metric. This implementation can not guarantee such maximality. I may have misunderstood the algorithm, but I can not see how it can be anything but a heuristic. Please contact me at sundsdal@gmail.com if you can provide python code that can guarantee maximality. Several optimizations are included in this code and it may be hard to read. Commented code to come.

A POSSIBLE ALTERNATIVE:

For an 'unconstrained' graph, that is one they describe as having the sum of the degree sequence be even(ie all undirected graphs) they present a simpler algorithm. It is as follows

> "For each vertex i: if di is even then attach di/2 self-loops; if di is odd, then attach (di-1)/2 self-loops, leaving one available "stub". Second for all remaining vertices with "stubs" connect them in pairs according to decreasing values of di."[1]

Since this only works for undirected graphs anyway, perhaps this is the better method? Note this also returns a graph with a larger s_metric than the other method, and it seems to have the same degree sequence, though I haven't tested it extensively.

6.6 Directed

Generators for some directed graphs.

gn_graph: growing network gnc_graph: growing network with copying gnr_graph: growing network with redirection scale_free_graph: scale free directed graph

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

6.6.1 networkx.generators.directed.gn_graph

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

[R66]

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 probabiliy p the link is instead "redirected" to the successor node of the target. The graph is always a (directed) tree.

Parameters n : int

The number of nodes for the generated graph.

p : float

The redirection probability.

create_using : graph, optional (default DiGraph)

Return graph of this type. The instance will be cleared.

seed : hashable object, optional

The seed for the random number generator.

References

[R68]

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

[R67]

6.6.4 networkx.generators.directed.scale_free_graph

scale_free_graph(*n, alpha=0.40999999999999998, beta=0.54000000000000004, gamma=0.050000000000000003, delta_in=0.20000000000000001, delta_out=0, create_using=None, seed=None*)

Return a scale free directed graph.

Parameters **n** : integer

Number of nodes in graph

alpha : float

Probability for adding a new node connected to an existing node chosen randomly according to the in-degree distribution.

beta : float

Probability for adding an edge between two existing nodes. One existing node is chosen randomly according the in-degree distribution and the other chosen randomly according to the out-degree distribution.

gamma : float

Probability for adding a new node conecgted to an existing node chosen randomly according to the out-degree distribution.

delta_in : float

Bias for choosing ndoes from in-degree distribution.

delta_out : float

Bias for choosing ndoes from out-degree distribution.

create_using : graph, optional (default MultiDiGraph)

Use this graph instance to start the process (default=3-cycle).

seed : integer, optional

Seed for random number generator

Notes

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

References

[R69]

Examples

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

6.7 Geometric

Generators for geometric graphs.

random_geometric_graph(n, radius[, ...])	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

kl_connected_subgraph(G, k, l[, low_memory, ...])	Returns the maximum locally (k,l) connected subgraph of G.
is_kl_connected(G, k, l[, low_memory])	Returns True if G is kl connected.

6.8.1 networkx.generators.hybrid.kl_connected_subgraph

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.

bipartite_configuration_model(aseq, bseq[, ...])	Return a random bipartite graph from two given degree sequences.
bipartite_havel_hakimi_graph(aseq, bseq[, ...])	Return a bipartite graph from two given degree sequences using a Havel-Hakimi style construction.
bipartite_reverse_havel_hakimi_graph(aseq, bseq)	Return a bipartite graph from two given degree sequences using a Havel-Hakimi style construction.
bipartite_alternating_havel_hakimi_graph(...)	Return a bipartite graph from two given degree sequences using a alternating Havel-Hakimi style construction.
bipartite_preferential_attachment_graph(aseq, p)	Create a bipartite graph with a preferential attachment model from a given single degree sequence.
bipartite_random_regular_graph(d, n[, ...])	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: sum(aseq)=sum(bseq) If no graph type is specified use MultiGraph with parallel edges. If you want a graph with no parallel edges use create_using=Graph() but then the resulting degree sequences might not be exact.

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: sum(aseq)=sum(bseq) If no graph type is specified use MultiGraph with parallel edges. If you want a graph with no parallel edges use create_using=Graph() but then the resulting degree sequences might not be exact.

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: sum(aseq)=sum(bseq) If no graph type is specified use MultiGraph with parallel edges. If you want a graph with no parallel edges use create_using=Graph() but then the resulting degree sequences might not be exact.

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 a alternating Havel-Hakimi style construction.

Parameters **aseq** : list or iterator

Degree sequence for node set A.

bseq : list or iterator

Degree sequence for node set B.

create_using : NetworkX graph instance, optional

Return graph of this type.

Nodes from the set A are connected to nodes in the set B by :

connecting the highest degree nodes in set A to :

alternatively the highest and the lowest degree nodes in set :

B until all stubs are connected. :

Notes

The sum of the two sequences must be equal: sum(aseq)=sum(bseq) If no graph type is specified use MultiGraph with parallel edges. If you want a graph with no parallel edges use create_using=Graph() but then the resulting degree sequences might not be exact.

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>=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<=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

Stocastic 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

CHAPTER

SEVEN

LINEAR ALGEBRA

7.1 Spectrum

Laplacian, adjacency matrix, and spectrum of graphs.

adj_matrix(G[, nodelist])	Return adjacency matrix of G.
laplacian(G[, nodelist])	Return the Laplacian matrix of G.
normalized_laplacian(G[, nodelist])	Return the normalized Laplacian matrix of G.
laplacian_spectrum(G)	Return eigenvalues of the Laplacian of G
adjacency_spectrum(G)	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

[R92]

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.

attr_matrix(G[, edge_attr, node_attr, ...])	Returns a NumPy matrix using attributes from G.
attr_sparse_matrix(G[, edge_attr, ...])	Returns a SciPy sparse matrix using attributes from G.

7.2.1 networkx.linalg.attrmatrix.attr_matrix

attr_matrix(*G, edge_attr=None, node_attr=None, normalized=False, rc_order=None, dtype=None, order=None*)

Returns a NumPy matrix using attributes from G.

If only *G* is passed in, then the adjacency matrix is constructed.

Let A be a discrete set of values for the node attribute *node_attr*. Then the elements of A represent the rows and columns of the constructed matrix. Now, iterate through every edge e=(u,v) in *G* and consider the value of the edge attribute *edge_attr*. If ua and va are the values of the node attribute *node_attr* for u and v, respectively, then the value of the edge attribute is added to the matrix element at (ua, va).

Parameters G : graph

The NetworkX graph used to construct the NumPy matrix.

edge_attr : str, optional

Each element of the matrix represents a running total of the specified edge attribute for edges whose node attributes correspond to the rows/cols of the matirx. The attribute must be present for all edges in the graph. If no attribute is specified, then we just count the number of edges whose node attributes correspond to the matrix element.

node_attr : str, optional

Each row and column in the matrix represents a particular value of the node attribute. The attribute must be present for all nodes in the graph. Note, the values of this attribute should be reliably hashable. So, float values are not recommended. If no attribute is specified, then the rows and columns will be the nodes of the graph.

normalized : bool, optional

If True, then each row is normalized by the summation of its values.

rc_order : list, optional

A list of the node attribute values. This list specifies the ordering of rows and columns of the array. If no ordering is provided, then the ordering will be random (and also, a return value).

Returns M : NumPy matrix

The attribute matrix.

ordering : list

If *rc_order* was specified, then only the matrix is returned. However, if *rc_order* was None, then the ordering used to construct the matrix is returned as well.

Examples

Construct an adjacency matrix:

```
>>> G = nx.Graph()
>>> G.add_edge(0,1,thickness=1,weight=3)
>>> G.add_edge(0,2,thickness=2)
>>> G.add_edge(1,2,thickness=3)
>>> nx.attr_matrix(G, rc_order=[0,1,2])
matrix([[ 0.,  1.,  1.],
        [ 1.,  0.,  1.],
        [ 1.,  1.,  0.]])
```

Alternatively, we can obtain the matrix describing edge thickness.

```
>>> nx.attr_matrix(G, edge_attr='thickness', rc_order=[0,1,2])
matrix([[ 0.,  1.,  2.],
        [ 1.,  0.,  3.],
        [ 2.,  3.,  0.]])
```

We can also color the nodes and ask for the probability distribution over all edges (u,v) describing:

Pr(v has color Y | u has color X)

```
>>> G.node[0]['color'] = 'red'
>>> G.node[1]['color'] = 'red'
>>> G.node[2]['color'] = 'blue'
>>> rc = ['red', 'blue']
>>> nx.attr_matrix(G, node_attr='color', normalized=True, rc_order=rc)
matrix([[ 0.33333333,  0.66666667],
        [ 1.        ,  0.        ]])
```

For example, the above tells us that for all edges (u,v):

Pr(v is red | u is red) = 1/3 Pr(v is blue | u is red) = 2/3

Pr(v is red | u is blue) = 1 Pr(v is blue | u is blue) = 0

Finally, we can obtain the total weights listed by the node colors.

```
>>> nx.attr_matrix(G, edge_attr='weight', node_attr='color', rc_order=rc)
matrix([[ 3.,  2.],
        [ 2.,  0.]])
```

Thus, the total weight over all edges (u,v) with u and v having colors:

(red, red) is 3 # the sole contribution is from edge (0,1) (red, blue) is 2 # contributions from edges (0,2) and (1,2) (blue, red) is 2 # same as (red, blue) since graph is undirected (blue, blue) is 0 # there are no edges with blue endpoints

7.2.2 networkx.linalg.attrmatrix.attr_sparse_matrix

attr_sparse_matrix(*G, edge_attr=None, node_attr=None, normalized=False, rc_order=None, dtype=None*)

Returns a SciPy sparse matrix using attributes from G.

If only *G* is passed in, then the adjacency matrix is constructed.

Let A be a discrete set of values for the node attribute *node_attr*. Then the elements of A represent the rows and columns of the constructed matrix. Now, iterate through every edge e=(u,v) in *G* and consider the value of the edge attribute *edge_attr*. If ua and va are the values of the node attribute *node_attr* for u and v, respectively, then the value of the edge attribute is added to the matrix element at (ua, va).

Parameters G : graph

The NetworkX graph used to construct the NumPy matrix.

edge_attr : str, optional

Each element of the matrix represents a running total of the specified edge attribute for edges whose node attributes correspond to the rows/cols of the matirx. The attribute must be present for all edges in the graph. If no attribute is specified, then we just count the number of edges whose node attributes correspond to the matrix element.

node_attr : str, optional

Each row and column in the matrix represents a particular value of the node attribute. The attribute must be present for all nodes in the graph. Note, the values of this attribute should be reliably hashable. So, float values are not recommended. If no attribute is specified, then the rows and columns will be the nodes of the graph.

normalized : bool, optional

If True, then each row is normalized by the summation of its values.

rc_order : list, optional

A list of the node attribute values. This list specifies the ordering of rows and columns of the array. If no ordering is provided, then the ordering will be random (and also, a return value).

Returns M : SciPy sparse matrix

The attribute matrix.

ordering : list

If *rc_order* was specified, then only the matrix is returned. However, if *rc_order* was None, then the ordering used to construct the matrix is returned as well.

Examples

Construct an adjacency matrix:

```
>>> G = nx.Graph()
>>> G.add_edge(0,1,thickness=1,weight=3)
>>> G.add_edge(0,2,thickness=2)
>>> G.add_edge(1,2,thickness=3)
>>> M = nx.attr_sparse_matrix(G, rc_order=[0,1,2])
>>> M.todense()
matrix([[ 0.,  1.,  1.],
        [ 1.,  0.,  1.],
        [ 1.,  1.,  0.]])
```

Alternatively, we can obtain the matrix describing edge thickness.

```
>>> M = nx.attr_sparse_matrix(G, edge_attr='thickness', rc_order=[0,1,2])
>>> M.todense()
matrix([[ 0.,  1.,  2.],
        [ 1.,  0.,  3.],
        [ 2.,  3.,  0.]])
```

We can also color the nodes and ask for the probability distribution over all edges (u,v) describing:

Pr(v has color Y | u has color X)

```
>>> G.node[0]['color'] = 'red'
>>> G.node[1]['color'] = 'red'
>>> G.node[2]['color'] = 'blue'
>>> rc = ['red', 'blue']
>>> M = nx.attr_sparse_matrix(G, node_attr='color',                          normalized
>>> M.todense()
matrix([[ 0.33333333,  0.66666667],
        [ 1.        ,  0.        ]])
```

For example, the above tells us that for all edges (u,v):

Pr(v is red | u is red) = 1/3 Pr(v is blue | u is red) = 2/3

Pr(v is red | u is blue) = 1 Pr(v is blue | u is blue) = 0

Finally, we can obtain the total weights listed by the node colors.

```
>>> M = nx.attr_sparse_matrix(G, edge_attr='weight',                          node_attr=
>>> M.todense()
matrix([[ 3.,  2.],
        [ 2.,  0.]])
```

Thus, the total weight over all edges (u,v) with u and v having colors:

(red, red) is 3 # the sole contribution is from edge (0,1) (red, blue) is 2 # contributions from edges (0,2) and (1,2) (blue, red) is 2 # same as (red, blue) since graph is undirected (blue, blue) is 0 # there are no edges with blue endpoints

CHAPTER

EIGHT

CONVERTING TO AND FROM OTHER DATA FORMATS

8.1 To NetworkX Graph

This module provides functions to convert NetworkX graphs to and from other formats.

The preferred way of converting data to a NetworkX graph is through the graph constuctor. The constructor calls the to_networkx_graph() function which attempts to guess the input type and convert it automatically.

8.1.1 Examples

Create a 10 node random graph from a numpy matrix

```
>>> import numpy
>>> a=numpy.reshape(numpy.random.random_integers(0,1,size=100),(10,10))
>>> D=nx.DiGraph(a)
```

or equivalently

```
>>> D=nx.to_networkx_graph(a,create_using=nx.DiGraph())
```

Create a graph with a single edge from a dictionary of dictionaries

```
>>> d={0: {1: 1}} # dict-of-dicts single edge (0,1)
>>> G=nx.Graph(d)
```

8.1.2 See Also

nx_pygraphviz, nx_pydot

`to_networkx_graph`(data[, create_using, ...])	Make a NetworkX graph from a known data structure.

8.1.3 networkx.convert.to_networkx_graph

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

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

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(),"abcdefghijklmnopqrstuvwxyz"))
>>> H=nx.relabel_nodes(G,mapping) # nodes a..z
>>> mapping=dict(zip(G.nodes(),range(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

to_dict_of_dicts(G[, nodelist, edge_data])	Return adjacency representation of graph as a dictionary of dictionaries.
from_dict_of_dicts(d[, create_using, ...])	Return a graph from a dictionary of dictionaries.

8.3.1 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 edgedata is None, the edgedata in G is used to fill the values. If G is a multigraph, the edgedata is a dict for each pair (u,v).

8.3.2 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

to_dict_of_lists(G[, nodelist])	Return adjacency representation of graph as a dictionary of lists.
from_dict_of_lists(d[, create_using])	Return a graph from a dictionary of lists.
to_edgelist(G[, nodelist])	Return a list of edges in the graph.
from_edgelist(edgelist[, create_using])	Return a graph from a list of edges.

8.4.1 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

to_numpy_matrix(G[, nodelist, dtype, order])	Return the graph adjacency matrix as a NumPy matrix.
from_numpy_matrix(A[, create_using])	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

to_scipy_sparse_matrix(G[, nodelist, dtype])	Return the graph adjacency matrix as a SciPy sparse matrix.
from_scipy_sparse_matrix(A[, create_using])	Return a graph from scipy sparse matrix adjacency list.

8.6.1 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.

nodelist : list, optional

The rows and columns are ordered according to the nodes in *nodelist*. If *nodelist* is None, then the ordering is produced by G.nodes().

dtype : NumPy data-type, optional

A valid NumPy dtype used to initialize the array. If None, then the NumPy default is used.

Returns **M** : SciPy sparse matrix

Graph adjacency matrix.

Notes

The matrix entries are populated using the 'weight' edge attribute. When an edge does not have the 'weight' attribute, the value of the entry is 1. For multiple edges, the 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*.

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)
```

CHAPTER

NINE

READING AND WRITING GRAPHS

9.1 Adjacency List

Read and write NetworkX graphs as adjacency lists.

Adjacency list format is useful for graphs without data associated with nodes or edges and for nodes that can be meaningfully represented as strings.

9.1.1 Format

The adjacency list format consists of lines with node labels. The first label in a line is the source node. Further labels in the line are considered target nodes and are added to the graph along with an edge between the source node and target node.

The graph with edges a-b, a-c, d-e can be represented as the following adjacency list (anything following the # in a line is a comment):

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

read_adjlist(path[, comments, delimiter, ...])	Read graph in adjacency list format from path.
write_adjlist(G, path[, comments, ...])	Write graph G in single-line adjacency-list format to path.
parse_adjlist(lines[, comments, delimiter, ...])	Parse lines of a graph adjacency list representation.
generate_adjlist(G[, delimiter])	Generate a single line of the graph G in adjacency list format.

9.1.2 networkx.read_adjlist

read_adjlist(*path, comments='#', delimiter=' ', create_using=None, nodetype=None, encoding='utf-8'*)

Read graph in adjacency list format from path.

Parameters **path** : string or file

Filename or file handle to read. Filenames ending in .gz or .bz2 will be uncompressed.

create_using: NetworkX graph container :

Use given NetworkX graph for holding nodes or edges.

nodetype : Python type, optional

Convert nodes to this type.

comments : string, optional

Marker for comment lines

delimiter : string, optional

Separator for node labels

create_using: NetworkX graph container :

Use given NetworkX graph for holding nodes or edges.

Returns G: NetworkX graph :

The graph corresponding to the lines in adjacency list format.

See Also:

write_adjlist

Notes

This format does not store graph or node data.

Examples

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

The path can be a filehandle or a string with the name of the file. If a filehandle is provided, it has to be opened in 'rb' mode.

```
>>> fh=open("test.adjlist", 'rb')
>>> G=nx.read_adjlist(fh)
```

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

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

The optional nodetype is a function to convert node strings to nodetype.

For example

```
>>> G=nx.read_adjlist("test.adjlist", nodetype=int)
```

will attempt to convert all nodes to integer type.

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

The optional create_using parameter is a NetworkX graph container. The default is Graph(), an undirected graph. To read the data as a directed graph use

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

9.1.3 networkx.write_adjlist

write_adjlist (*G, path, comments='#', delimiter=' ', encoding='utf-8'*)
Write graph G in single-line adjacency-list format to path.

Parameters G : NetworkX graph

path : string or file

Filename or file handle for data output. Filenames ending in .gz or .bz2 will be compressed.

comments : string, optional

Marker for comment lines

delimiter : string, optional

Separator for node labels

encoding : string, optional

Text encoding.

See Also:

read_adjlist, generate_adjlist

Notes

This format does not store graph, node, or edge data.

Examples

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

The path can be a filehandle or a string with the name of the file. If a filehandle is provided, it has to be opened in 'wb' mode.

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

9.1.4 networkx.parse_adjlist

parse_adjlist (*lines, comments='#', delimiter=' ', create_using=None, nodetype=None*)
Parse lines of a graph adjacency list representation.

Parameters lines : list or iterator of strings

Input data in adjlist format

create_using: NetworkX graph container :

Use given NetworkX graph for holding nodes or edges.

nodetype : Python type, optional

Convert nodes to this type.

comments : string, optional

Marker for comment lines

delimiter : string, optional

Separator for node labels

create_using: NetworkX graph container :

Use given NetworkX graph for holding nodes or edges.

Returns G: NetworkX graph :

The graph corresponding to the lines in adjacency list format.

See Also:

read_adjlist

Examples

```
>>> lines = ['1 2 5',
...          '2 3 4',
...          '3 5',
...          '4',
...          '5']
>>> G = nx.parse_adjlist(lines, nodetype = int)
>>> G.nodes()
[1, 2, 3, 4, 5]
>>> G.edges()
[(1, 2), (1, 5), (2, 3), (2, 4), (3, 5)]
```

9.1.5 networkx.generate_adjlist

generate_adjlist(*G, delimiter=' '*)

Generate a single line of the graph G in adjacency list format.

Parameters G : NetworkX graph

delimiter : string, optional

Separator for node labels

See Also:

write_adjlist, read_adjlist

Examples

```
>>> G = nx.lollipop_graph(4, 3)
>>> for line in nx.generate_adjlist(G):
...     print(line)
0 1 2 3
1 2 3
2 3
3 4
```

```
4 5
5 6
6
```

9.2 Edge List

Read and write NetworkX graphs as edge lists.

The multi-line adjacency list format is useful for graphs with nodes that can be meaningfully represented as strings. With the edgelist format simple edge data can be stored but node or graph data is not. There is no way of representing isolated nodes unless the node has a self-loop edge.

9.2.1 Format

You can read or write three formats of edge lists with these functions.

Node pairs with no data:

```
1 2
```

Python dictionary as data:

```
1 2 {'weight':7, 'color':'green'}
```

Arbitrary data:

```
1 2 7 green
```

`read_edgelist`(path[, comments, delimiter, ...])	Read a graph from a list of edges.
`write_edgelist`(G, path[, comments, ...])	Write graph as a list of edges.
`read_weighted_edgelist`(path[, comments, ...])	Read a graph as list of edges with numeric weights.
`write_weighted_edgelist`(G, path[, comments, ...])	Write graph G as a list of edges with numeric weights.
`generate_edgelist`(G[, delimiter, data])	Generate a single line of the graph G in edge list format.
`parse_edgelist`(lines[, comments, delimiter, ...])	Parse lines of an edge list representation of a graph.

9.2.2 networkx.read_edgelist

read_edgelist (*path, comments='#', delimiter=' ', create_using=None, nodetype=None, data=True, edgetype=None, encoding='utf-8'*)

Read a graph from a list of edges.

Parameters path : file or string

File or filename to write. If a file is provided, it must be opened in 'rb' mode. Filenames ending in .gz or .bz2 will be uncompressed.

comments : string, optional

The character used to indicate the start of a comment.

delimiter : string, optional

The string used to separate values. The default is whitespace.

create_using : Graph container, optional, Use specified container to build graph. The default is networkx.Graph, an undirected graph.

nodetype : int, float, str, Python type, optional

Convert node data from strings to specified type

data : bool or list of (label,type) tuples

Tuples specifying dictionary key names and types for edge data

edgetype : int, float, str, Python type, optional OBSOLETE

Convert edge data from strings to specified type and use as 'weight'

encoding: string, optional :

Specify which encoding to use when reading file.

Returns G : graph

A networkx Graph or other type specified with create_using

See Also:

parse_edgelist

Notes

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

Examples

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

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

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

See parse_edgelist() for more examples of formatting.

9.2.3 networkx.write_edgelist

write_edgelist (*G, path, comments='#', delimiter=' ', data=True, encoding='utf-8'*)

Write graph as a list of edges.

Parameters G : graph

A NetworkX graph

path : file or string

File or filename to write. If a file is provided, it must be opened in 'wb' mode. Filenames ending in .gz or .bz2 will be compressed.

comments : string, optional

The character used to indicate the start of a comment

delimiter : string, optional

The string used to separate values. The default is whitespace.

data : bool or list, optional

If False write no edge data. If True write a string representation of the edge data dictionary.. If a list (or other iterable) is provided, write the keys specified in the list.

encoding: string, optional :

Specify which encoding to use when writing file.

See Also:

write_edgelist, write_weighted_edgelist

Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_edgelist(G, "test.edgelist")
>>> G=nx.path_graph(4)
>>> fh=open("test.edgelist",'wb')
>>> nx.write_edgelist(G, fh)
>>> nx.write_edgelist(G, "test.edgelist.gz")
>>> nx.write_edgelist(G, "test.edgelist.gz", data=False)
```

```
>>> G=nx.Graph()
>>> G.add_edge(1,2,weight=7,color='red')
>>> nx.write_edgelist(G,'test.edgelist',data=False)
>>> nx.write_edgelist(G,'test.edgelist',data=['color'])
>>> nx.write_edgelist(G,'test.edgelist',data=['color','weight'])
```

9.2.4 networkx.read_weighted_edgelist

read_weighted_edgelist (*path, comments='#', delimiter=' ', create_using=None, nodetype=None, encoding='utf-8'*)

Read a graph as list of edges with numeric weights.

Parameters path : file or string

File or filename to write. If a file is provided, it must be opened in 'rb' mode. Filenames ending in .gz or .bz2 will be uncompressed.

comments : string, optional

The character used to indicate the start of a comment.

delimiter : string, optional

The string used to separate values. The default is whitespace.

create_using : Graph container, optional,

Use specified container to build graph. The default is networkx.Graph, an undirected graph.

nodetype : int, float, str, Python type, optional

Convert node data from strings to specified type

encoding: string, optional :

Specify which encoding to use when reading file.

Returns G : graph

A networkx Graph or other type specified with create_using

Notes

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

Example edgelist file format.

With numeric edge data:

```
# read with
# >>> G=nx.read_weighted_edgelist(fh)
# source target data
a b 1
a c 3.14159
d e 42
```

9.2.5 networkx.write_weighted_edgelist

write_weighted_edgelist (*G, path, comments='#', delimiter=' ', encoding='utf-8'*)

Write graph G as a list of edges with numeric weights.

Parameters G : graph

A NetworkX graph

path : file or string

File or filename to write. If a file is provided, it must be opened in 'wb' mode. Filenames ending in .gz or .bz2 will be compressed.

comments : string, optional

The character used to indicate the start of a comment

delimiter : string, optional

The string used to separate values. The default is whitespace.

encoding: string, optional :

Specify which encoding to use when writing file.

See Also:

read_edgelist, write_edgelist, write_weighted_edgelist

Examples

```
>>> G=nx.Graph()
>>> G.add_edge(1,2,weight=7)
>>> nx.write_weighted_edgelist(G, 'weighted.edgelist')
```

9.2.6 networkx.generate_edgelist

generate_edgelist (*G, delimiter=' ', data=True*)

Generate a single line of the graph G in edge list format.

Parameters G : NetworkX graph

delimiter : string, optional

Separator for node labels

data : bool or list of keys

If False generate no edge data. If True use a dictionary representation of edge data. If a list of keys use a list of data values corresponding to the keys.

See Also:

write_adjlist, read_adjlist

Examples

```
>>> G = nx.lollipop_graph(4, 3)
>>> G[1][2]['weight'] = 3
>>> G[3][4]['capacity'] = 12
>>> for line in nx.generate_edgelist(G, data=False):
...     print(line)
0 1
0 2
0 3
1 2
1 3
2 3
3 4
4 5
5 6
```

```
>>> for line in nx.generate_edgelist(G):
...     print(line)
0 1 {}
0 2 {}
0 3 {}
1 2 {'weight': 3}
1 3 {}
2 3 {}
3 4 {'capacity': 12}
4 5 {}
5 6 {}
```

```
>>> for line in nx.generate_edgelist(G,data=['weight']):
...     print(line)
0 1
0 2
0 3
1 2 3
1 3
2 3
3 4
4 5
5 6
```

9.2.7 networkx.parse_edgelist

parse_edgelist (*lines, comments='#', delimiter=' ', create_using=None, nodetype=None, data=True*)
Parse lines of an edge list representation of a graph.

Returns G: NetworkX Graph :

The graph corresponding to lines

data : bool or list of (label,type) tuples

If False generate no edge data or if True use a dictionary representation of edge data or a list tuples specifying dictionary key names and types for edge data.

create_using: NetworkX graph container, optional :

Use given NetworkX graph for holding nodes or edges.

nodetype : Python type, optional

Convert nodes to this type.

comments : string, optional

Marker for comment lines

delimiter : string, optional

Separator for node labels

create_using: NetworkX graph container :

Use given NetworkX graph for holding nodes or edges.

See Also:

read_weighted_edgelist

Examples

Edgelist with no data:

```
>>> lines = ["1 2",
...          "2 3",
...          "3 4"]
>>> G = nx.parse_edgelist(lines, nodetype = int)
>>> G.nodes()
```

```
[1, 2, 3, 4]
>>> G.edges()
[(1, 2), (2, 3), (3, 4)]
```

Edgelist with data in Python dictionary representation:

```
>>> lines = ["1 2 {'weight':3}",
...          "2 3 {'weight':27}",
...          "3 4 {'weight':3.0}"]
>>> G = nx.parse_edgelist(lines, nodetype = int)
>>> G.nodes()
[1, 2, 3, 4]
>>> G.edges(data = True)
[(1, 2, {'weight': 3}), (2, 3, {'weight': 27}), (3, 4, {'weight': 3.0})]
```

Edgelist with data in a list:

```
>>> lines = ["1 2 3",
...          "2 3 27",
...          "3 4 3.0"]
>>> G = nx.parse_edgelist(lines, nodetype = int, data=(('weight',float),))
>>> G.nodes()
[1, 2, 3, 4]
>>> G.edges(data = True)
[(1, 2, {'weight': 3.0}), (2, 3, {'weight': 27.0}), (3, 4, {'weight': 3.0})]
```

9.3 GML

Read graphs in GML format.

"GML, the G>raph Modelling Language, is our proposal for a portable file format for graphs. GML's key features are portability, simple syntax, extensibility and flexibility. A GML file consists of a hierarchical key-value lists. Graphs can be annotated with arbitrary data structures. The idea for a common file format was born at the GD'95; this proposal is the outcome of many discussions. GML is the standard file format in the Graphlet graph editor system. It has been overtaken and adapted by several other systems for drawing graphs."

See http://www.infosun.fim.uni-passau.de/Graphlet/GML/gml-tr.html

Requires pyparsing: http://pyparsing.wikispaces.com/

9.3.1 Format

See http://www.infosun.fim.uni-passau.de/Graphlet/GML/gml-tr.html for format specification.

Example graphs in GML format: http://www-personal.umich.edu/~mejn/netdata/

read_gml(path[, encoding])	Read graph in GML format from path.
write_gml(G, path)	Write the graph G in GML format to the file or file handle path.
parse_gml(lines)	Parse GML graph from a string or iterable.
generate_gml(G)	Generate a single entry of the graph G in GML format.

9.3.2 networkx.read_gml

read_gml (*path, encoding='UTF-8'*)

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 pyparsing 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 pyparsing: http://pyparsing.wikispaces.com/

References

GML specification: http://www.infosun.fim.uni-passau.de/Graphlet/GML/gml-tr.html

Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_gml(G,'test.gml')
>>> H=nx.read_gml('test.gml')
```

9.3.3 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

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")
```

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

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

9.3.4 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

9.3.5 networkx.generate_gml

generate_gml(*G*)
Generate a single entry of the graph G in GML format.

Parameters G : NetworkX graph

9.4 Pickle

Read and write NetworkX graphs as Python pickles.

"The pickle module implements a fundamental, but powerful algorithm for serializing and de-serializing a Python object structure. "Pickling" is the process whereby a Python object hierarchy is converted into a byte stream, and "unpickling" is the inverse operation, whereby a byte stream is converted back into an object hierarchy."

Note that NetworkX graphs can contain any hashable Python object as node (not just integers and strings). For arbitrary data types it may be difficult to represent the data as text. In that case using Python pickles to store the graph data can be used.

9.4.1 Format

See http://docs.python.org/library/pickle.html

read_gpickle(path)	Read graph object in Python pickle format.
write_gpickle(G, path)	Write graph in Python pickle format.

9.4.2 networkx.read_gpickle

read_gpickle(*path*)

Read graph object in Python pickle format.

Pickles are a serialized byte stream of a Python object [R102]. This format will preserve Python objects used as nodes or edges.

Parameters **path** : file or string

File or filename to write. Filenames ending in .gz or .bz2 will be uncompressed.

Returns **G** : graph

A NetworkX graph

References

[R102]

Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_gpickle(G,"test.gpickle")
>>> G=nx.read_gpickle("test.gpickle")
```

9.4.3 networkx.write_gpickle

write_gpickle(*G, path*)

Write graph in Python pickle format.

Pickles are a serialized byte stream of a Python object [R106]. This format will preserve Python objects used as nodes or edges.

Parameters **G** : graph

A NetworkX graph

path : file or string

File or filename to write. Filenames ending in .gz or .bz2 will be compressed.

References

[R106]

Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_gpickle(G,"test.gpickle")
```

9.5 GraphML

Read and write graphs in GraphML format.

This implementation does not support mixed graphs (directed and unidirected edges together), hyperedges, nested graphs, or ports.

"GraphML is a comprehensive and easy-to-use file format for graphs. It consists of a language core to describe the structural properties of a graph and a flexible extension mechanism to add application-specific data. Its main features include support of

- directed, undirected, and mixed graphs,
- hypergraphs,
- hierarchical graphs,
- graphical representations,
- references to external data,
- application-specific attribute data, and
- light-weight parsers.

Unlike many other file formats for graphs, GraphML does not use a custom syntax. Instead, it is based on XML and hence ideally suited as a common denominator for all kinds of services generating, archiving, or processing graphs."

http://graphml.graphdrawing.org/

9.5.1 Format

GraphML is an XML format. See http://graphml.graphdrawing.org/specification.html for the specification and http://graphml.graphdrawing.org/primer/graphml-primer.html for examples.

read_graphml(path[, node_type])	Read graph in GraphML format from path.
write_graphml(G, path[, encoding])	Write G in GraphML XML format to path

9.5.2 networkx.read_graphml

read_graphml (*path, node_type=<type 'str'>*)

Read graph in GraphML format from path.

Parameters path : file or string

File or filename to write. Filenames ending in .gz or .bz2 will be compressed.

Returns graph: NetworkX graph :

If no parallel edges are found a Graph or DiGraph is returned. Otherwise a MultiGraph or MultiDiGraph is returned.

Notes

This implementation does not support mixed graphs (directed and unidirected edges together), hypergraphs, nested graphs, or ports.

9.5.3 networkx.write_graphml

write_graphml (*G, path, encoding='utf-8'*)
Write G in GraphML XML format to path

Parameters G : graph

A networkx graph

path : file or string

File or filename to write. Filenames ending in .gz or .bz2 will be compressed.

Notes

This implementation does not support mixed graphs (directed and unidirected edges together) hyperedges, nested graphs, or ports.

Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_graphml(G, "test.graphml")
```

9.6 LEDA

Read graphs in LEDA format.

LEDA is a C++ class library for efficient data types and algorithms.

9.6.1 Format

See http://www.algorithmic-solutions.info/leda_guide/graphs/leda_native_graph_fileformat.html

read_leda(path[, encoding])	Read graph in LEDA format from path.
parse_leda(lines)	Read graph in LEDA format from string or iterable.

9.6.2 networkx.read_leda

read_leda (*path, encoding='UTF-8'*)

Read graph in LEDA format from path.

Parameters **path** : file or string

File or filename to read. Filenames ending in .gz or .bz2 will be uncompressed.

Returns **G** : NetworkX graph

References

[R103]

Examples

G=nx.read_leda('file.leda')

9.6.3 networkx.parse_leda

parse_leda (*lines*)

Read graph in LEDA format from string or iterable.

Parameters **lines** : string or iterable

Data in LEDA format.

Returns **G** : NetworkX graph

References

[R101]

Examples

G=nx.parse_leda(string)

9.7 YAML

Read and write NetworkX graphs in YAML format.

"YAML is a data serialization format designed for human readability and interaction with scripting languages." See http://www.yaml.org for documentation.

9.7.1 Format

http://pyyaml.org/wiki/PyYAML

read_yaml(path)	Read graph in YAML format from path.
write_yaml(G, path, **kwds[, encoding])	Write graph G in YAML format to path.

9.7.2 networkx.read_yaml

read_yaml (*path*)

Read graph in YAML format from path.

YAML is a data serialization format designed for human readability and interaction with scripting languages [R104].

Parameters path : file or string

File or filename to read. Filenames ending in .gz or .bz2 will be uncompressed.

Returns G : NetworkX graph

References

[R104]

Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_yaml(G,'test.yaml')
>>> G=nx.read_yaml('test.yaml')
```

9.7.3 networkx.write_yaml

write_yaml (*G, path, encoding='UTF-8', **kwds*)

Write graph G in YAML format to path.

YAML is a data serialization format designed for human readability and interaction with scripting languages [R107].

Parameters G : graph

A NetworkX graph

path : file or string

File or filename to write. Filenames ending in .gz or .bz2 will be compressed.

encoding: string, optional :

Specify which encoding to use when writing file.

References

[R107]

Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_yaml(G,'test.yaml')
```

9.8 SparseGraph6

Read graphs in graph6 and sparse6 format.

9.8.1 Format

"graph6 and sparse6 are formats for storing undirected graphs in a compact manner, using only printable ASCII characters. Files in these formats have text type and contain one line per graph." http://cs.anu.edu.au/~bdm/data/formats.html

See http://cs.anu.edu.au/~bdm/data/formats.txt for details.

read_graph6(path)	Read simple undirected graphs in graph6 format from path.
parse_graph6(str)	Read a simple undirected graph in graph6 format from string.
read_graph6_list(path)	Read simple undirected graphs in graph6 format from path.
read_sparse6(path)	Read simple undirected graphs in sparse6 format from path.
parse_sparse6(string)	Read undirected graph in sparse6 format from string.
read_sparse6_list(path)	Read undirected graphs in sparse6 format from path.

9.8.2 networkx.read_graph6

read_graph6(*path*)

Read simple undirected graphs in graph6 format from path.

Returns a single Graph.

9.8.3 networkx.parse_graph6

parse_graph6(*str*)

Read a simple undirected graph in graph6 format from string.

Returns a single Graph.

9.8.4 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.

9.8.5 networkx.read_sparse6

read_sparse6(*path*)

Read simple undirected graphs in sparse6 format from path.

Returns a single MultiGraph.

9.8.6 networkx.parse_sparse6

parse_sparse6 (*string*)

Read undirected graph in sparse6 format from string.

Returns a MultiGraph.

9.8.7 networkx.read_sparse6_list

read_sparse6_list (*path*)

Read undirected graphs in sparse6 format from path.

Returns a list of MultiGraphs, one for each line in file.

9.9 Pajek

Read graphs in Pajek format.

This implementation handles directed and undirected graphs including those with self loops and parallel edges.

9.9.1 Format

See http://vlado.fmf.uni-lj.si/pub/networks/pajek/doc/draweps.htm for format information.

read_pajek(path[, encoding])	Read graph in Pajek format from path.
write_pajek(G, path[, encoding])	Write graph in Pajek format to path.
parse_pajek(lines)	Parse Pajek format graph from string or iterable.

9.9.2 networkx.read_pajek

read_pajek (*path, encoding='UTF-8'*)

Read graph in Pajek format from path.

Parameters path : file or string

File or filename to write. Filenames ending in .gz or .bz2 will be uncompressed.

Returns G : NetworkX MultiGraph or MultiDiGraph.

References

See http://vlado.fmf.uni-lj.si/pub/networks/pajek/doc/draweps.htm for format information.

Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_pajek(G, "test.net")
>>> G=nx.read_pajek("test.net")
```

To create a Graph instead of a MultiGraph use

```
>>> G1=nx.Graph(G)
```

9.9.3 networkx.write_pajek

write_pajek(*G, path, encoding='UTF-8'*)

Write graph in Pajek format to path.

Parameters G : graph

A Networkx graph

path : file or string

File or filename to write. Filenames ending in .gz or .bz2 will be compressed.

References

See http://vlado.fmf.uni-lj.si/pub/networks/pajek/doc/draweps.htm for format information.

Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_pajek(G, "test.net")
```

9.9.4 networkx.parse_pajek

parse_pajek(*lines*)

Parse Pajek format graph from string or iterable.

Parameters lines : string or iterable

Data in Pajek format.

Returns G : NetworkX graph

See Also:

read_pajek

CHAPTER

TEN

DRAWING

10.1 Matplotlib

Draw networks with matplotlib (pylab).

10.1.1 See Also

matplotlib: http://matplotlib.sourceforge.net/

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

draw(G, **kwds[, pos, ax, hold])	Draw the graph G with Matplotlib (pylab).
draw_networkx(G, **kwds[, pos, with_labels])	Draw the graph G using Matplotlib.
draw_networkx_nodes(G, pos, **kwds[, ...])	Draw the nodes of the graph G.
draw_networkx_edges(G, pos, **kwds[, ...])	Draw the edges of the graph G.
draw_networkx_labels(G, pos, **kwds[, ...])	Draw node labels on the graph G.
draw_networkx_edge_labels(G, pos, **kwds[, ...])	Draw edge labels.
draw_circular(G, **kwargs)	Draw the graph G with a circular layout.
draw_random(G, **kwargs)	Draw the graph G with a random layout.
draw_spectral(G, **kwargs)	Draw the graph G with a spectral layout.
draw_spring(G, **kwargs)	Draw the graph G with a spring layout.
draw_shell(G, **kwargs)	Draw networkx graph with shell layout.
draw_graphviz(G, **kwargs[, prog])	Draw networkx graph with graphviz layout.

10.1.2 networkx.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, **kwds*)

Draw the graph G using Matplotlib.

Draw the graph with Matplotlib with options for node positions, labeling, titles, and many other drawing features. See draw() for simple drawing without labels or axes.

Parameters **G** : graph

A networkx graph

pos : dictionary, optional

A dictionary with nodes as keys and positions as values. If not specified a spring layout positioning will be computed. See networkx.layout for functions that compute node positions.

ax : Matplotlib Axes object, optional

Draw the graph in the specified Matplotlib axes.

with_labels: bool, optional :

Set to True (default) to draw labels on the nodes.

nodelist: list, optional :

Draw only specified nodes (default G.nodes())

edgelist: list :

Draw only specified edges(default=G.edges())

node_size: scalar or array :

Size of nodes (default=300). If an array is specified it must be the same length as nodelist.

node_color: color string, or array of floats :

Node color. Can be a single color format string (default='r'), or a sequence of colors with the same length as nodelist. If numeric values are specified they will be mapped to colors using the cmap and vmin,vmax parameters. See matplotlib.scatter for more details.

node_shape: string :

The shape of the node. Specification is as matplotlib.scatter marker, one of 'so^>v<dph8' (default='o').

alpha: float :

The node transparency (default=1.0)

cmap: Matplotlib colormap :

Colormap for mapping intensities of nodes (default=None)

vmin,vmax: floats :

Minimum and maximum for node colormap scaling (default=None)

width': float :

Line width of edges (default =1.0)

edge_color: color string, or array of floats :

Edge color. Can be a single color format string (default='r'), or a sequence of colors with the same length as edgelist. If numeric values are specified they will be mapped to colors using the edge_cmap and edge_vmin,edge_vmax parameters.

edge_ cmap: Matplotlib colormap :

Colormap for mapping intensities of edges (default=None)

edge_vmin,edge_vmax: floats :

Minimum and maximum for edge colormap scaling (default=None)

style: string :

Edge line style (default='solid') (solid|dashed|dotted,dashdot)

labels: dictionary :

Node labels in a dictionary keyed by node of text labels (default=None)

font_size: int :

Font size for text labels (default=12)

font_color: string :

Font color string (default='k' black)

font_weight: string :

Font weight (default='normal')

font_family: string :

Font family (default='sans-serif')

See Also:

`draw`, `draw_networkx_nodes`, `draw_networkx_edges`, `draw_networkx_labels`, `draw_networkx_edge_labels`

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, **kwds*)

Draw the edges of the graph G.

This draws only the edges of the graph G.

Parameters G : graph

A networkx graph

pos : dictionary

A dictionary with nodes as keys and positions as values. If not specified a spring layout positioning will be computed. See networkx.layout for functions that compute node positions.

ax : Matplotlib Axes object, optional

Draw the graph in the specified Matplotlib axes.

alpha: float :

The edge transparency (default=1.0)

width': float :

Line width of edges (default =1.0)

edge_color: color string, or array of floats :

Edge color. Can be a single color format string (default='r'), or a sequence of colors with the same length as edgelist. If numeric values are specified they will be mapped to colors using the edge_cmap and edge_vmin,edge_vmax parameters.

edge_ cmap: Matplotlib colormap :

Colormap for mapping intensities of edges (default=None)

edge_vmin,edge_vmax: floats :

Minimum and maximum for edge colormap scaling (default=None)

style: string :

Edge line style (default='solid') (solid|dashed|dotted,dashdot)

See Also:

draw, draw_networkx, draw_networkx_nodes, draw_networkx_labels, draw_networkx_edge_labels

Notes

For directed graphs, "arrows" (actually just thicker stubs) are drawn at the head end. Arrows can be turned off with keyword arrows=False. Yes, it is ugly but drawing proper arrows with Matplotlib this way is tricky.

Examples

```
>>> G=nx.dodecahedral_graph()
>>> edges=nx.draw_networkx_edges(G,pos=nx.spring_layout(G))
```

Also see the NetworkX drawing examples at http://networkx.lanl.gov/gallery.html

10.1.6 networkx.draw_networkx_labels

draw_networkx_labels(*G, pos, labels=None, font_size=12, font_color='k', font_family='sans-serif', font_weight='normal', alpha=1.0, ax=None, **kwds*)

Draw node labels on the graph G.

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

from_agraph(A[, create_using])	Return a NetworkX Graph or DiGraph from a PyGraphviz graph.
to_agraph(N)	Return a pygraphviz graph from a NetworkX graph N.
write_dot(G, path)	Write NetworkX graph G to Graphviz dot format on path.
read_dot(path)	Return a NetworkX graph from a dot file on path.
graphviz_layout(G[, prog, root, args])	Create node positions for G using Graphviz.
pygraphviz_layout(G[, prog, root, args])	Create node positions for G using Graphviz.

10.2.3 networkx.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

from_pydot(P)	Return a NetworkX graph from a Pydot graph.
to_pydot(N[, strict])	Return a pydot graph from a NetworkX graph N.
write_dot(G, path)	Write NetworkX graph G to Graphviz dot format on path.
read_dot(path)	Return a NetworkX graph from a dot file on path.
graphviz_layout(G[, prog, root, args])	Create node positions for G using Graphviz.
pydot_layout(G, **kwds[, prog, root])	Create node positions using Pydot and Graphviz.

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, **kwds*)
Create node positions using Pydot and Graphviz.

Returns a dictionary of positions keyed by node.

Examples

```
>>> G=nx.complete_graph(4)
>>> pos=nx.pydot_layout(G)
>>> pos=nx.pydot_layout(G,prog='dot')
```

10.4 Graph Layout

Node positioning algorithms for graph drawing.

circular_layout(G[, dim, scale])	Position nodes on a circle.
random_layout(G[, dim])	Position nodes uniformly at random in the unit square.
shell_layout(G[, nlist, dim, scale])	Position nodes in concentric circles.
spring_layout(G[, dim, pos, fixed, ...])	Position nodes using Fruchterman-Reingold force-directed algorithm.
spectral_layout(G[, dim, weighted, scale])	Position nodes using the eigenvectors of the graph Laplacian.

10.4.1 networkx.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*)
Position nodes uniformly at random in the unit square.

For every node, a position is generated by choosing each of dim coordinates uniformly at random on the interval [0.0, 1.0).

NumPy (http://scipy.org) is required for this function.

Parameters **G** : NetworkX graph

A position will be assigned to every node in G.

dim : int

Dimension of layout.

Returns **dict :** :

A dictionary of positions keyed by node

Examples

```
>>> G = nx.lollipop_graph(4, 3)
>>> pos = nx.random_layout(G)
```

10.4.3 networkx.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 unidrected graphs when positioning the nodes.

For larger graphs (>500 nodes) this will use the SciPy sparse eigenvalue solver (ARPACK).

Examples

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

CHAPTER

ELEVEN

EXCEPTIONS

Base exceptions and errors for NetworkX.

class `NetworkXException()`
Base class for exceptions in NetworkX.

class `NetworkXError()`
Exception for a serious error in NetworkX

class `NetworkXPointlessConcept()`
Harary, F. and Read, R. "Is the Null Graph a Pointless Concept?" In Graphs and Combinatorics Conference, George Washington University. New York: Springer-Verlag, 1973.

class `NetworkXAlgorithmError()`
Exception for unexpected termination of algorithms.

class `NetworkXUnfeasible()`
Exception raised by algorithms trying to solve a problem instance that has no feasible solution.

class `NetworkXNoPath()`
Exception for algorithms that should return a path when running on graphs where such a path does not exist.

class `NetworkXUnbounded()`
Exception raised by algorithms trying to solve a maximization or a minimization problem instance that is unbounded.

CHAPTER

TWELVE

UTILITIES

Helpers for NetworkX.

These are not imported into the base networkx namespace but can be accessed, for example, as

```
>>> import networkx
>>> networkx.utils.is_string_like('spam')
True
```

12.1 Helper functions

is_string_like(obj)	Check if obj is string.
flatten(obj[, result])	Return flattened version of (possibly nested) iterable object.
iterable(obj)	Return True if obj is iterable with a well-defined len().
is_list_of_ints(intlist)	Return True if list is a list of ints.
_get_fh(path[, mode])	Return a file handle for given path.

12.1.1 networkx.utils.is_string_like

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

pareto_sequence(n[, exponent])	Return sample sequence of length n from a Pareto distribution.
powerlaw_sequence(n[, exponent])	Return sample sequence of length n from a power law distribution.
uniform_sequence(n)	Return sample sequence of length n from a uniform distribution.
cumulative_distribution(distribution)	Return normalized cumulative distribution from discrete distribution.
discrete_sequence(n[, distribution, ...])	Return sample sequence of length n from a given discrete distribution or discrete cumulative distribution.

12.3.1 networkx.utils.pareto_sequence

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

scipy_pareto_sequence(n[, exponent])	Return sample sequence of length n from a Pareto distribution.
scipy_powerlaw_sequence(n[, exponent])	Return sample sequence of length n from a power law distribution.
scipy_poisson_sequence(n[, mu])	Return sample sequence of length n from a Poisson distribution.
scipy_uniform_sequence(n)	Return sample sequence of length n from a uniform distribution.
scipy_discrete_sequence(n[, distribution])	Return sample sequence of length n from a given discrete distribution.

12.4.1 networkx.utils.scipy_pareto_sequence

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

CHAPTER THIRTEEN

LICENSE

NetworkX is distributed with the BSD license.

CHAPTER

FOURTEEN

CITING

To cite NetworkX please use the following publication:

Aric A. Hagberg, Daniel A. Schult and Pieter J. Swart, "Exploring network structure, dynamics, and function using NetworkX", in Proceedings of the 7th Python in Science Conference (SciPy2008), Gäel Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11–15, Aug 2008

CHAPTER

FIFTEEN

CREDITS

NetworkX was originally written by Aric Hagberg, Dan Schult, and Pieter Swart with the help of many others.

Thanks to Guido van Rossum for the idea of using Python for implementing a graph data structure http://www.python.org/doc/essays/graphs.html

Thanks to David Eppstein for the idea of representing a graph G so that "for n in G" loops over the nodes in G and G[n] are node n's neighbors.

Thanks to all those who have improved NetworkX by contributing code, bug reports (and fixes), documentation, and input on design, featues, and the future of NetworkX.

Thanks especially to the following contributors.

- Katy Bold contributed the Karate Club graph
- Hernan Rozenfeld added dorogovtsev_goltsev_mendes_graph and did stress testing
- Brendt Wohlberg added examples from the Stanford GraphBase
- Jim Bagrow reported bugs in the search methods
- Holly Johnsen helped fix the path based centrality measures
- Arnar Flatberg fixed the graph laplacian routines
- Chris Myers suggested using None as a default datatype, suggested improvements for the IO routines, added grid generator index tuple labeling and associated routines, and reported bugs
- Joel Miller tested and improved the connected components methods 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, and ported all of NetworkX to Python3.

CHAPTER

SIXTEEN

GLOSSARY

dictionary FIXME

ebunch An iteratable container of edge tuples like a list, iterator, or file.

edge Edges are either two-tuples of nodes (u,v) or three tuples of nodes with an edge attribute dictionary (u,v,dict).

edge attribute Edges can have arbitrary Python objects assigned as attributes by using keyword/value pairs when adding an edge assigning to the G.edge[u][v] attribute dictionary for the specified edge u-v.

hashable An object is hashable if it has a hash value which never changes during its lifetime (it needs a __hash__() method), and can be compared to other objects (it needs an __eq__() or __cmp__() method). Hashable objects which compare equal must have the same hash value.

Hashability makes an object usable as a dictionary key and a set member, because these data structures use the hash value internally.

All of Python's immutable built-in objects are hashable, while no mutable containers (such as lists or dictionaries) are. Objects which are instances of user-defined classes are hashable by default; they all compare unequal, and their hash value is their id().

Definition from http://docs.python.org/glossary.html

nbunch An nbunch is any iterable container of nodes that is not itself a node in the graph. It can be an iterable or an iterator, e.g. a list, set, graph, file, etc..

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

node attribute Nodes can have arbitrary Python objects assigned as attributes by using keyword/value pairs when adding a node or assigning to the G.node[n] attribute dictionary for the specified node n.

BIBLIOGRAPHY

[R49] Patrick Doreian, Vladimir Batagelj, and Anuska Ferligoj "Generalized Blockmodeling",Cambridge University Press, 2004.

[R38] 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

[R39] 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

[R50] 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

[R51] 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

[R40] 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

[R41] A measure of betweenness centrality based on random walks, M. E. J. Newman, Social Networks 27, 39-54 (2005).

[R42] 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

[R43] A measure of betweenness centrality based on random walks, M. E. J. Newman, Social Networks 27, 39-54 (2005).

[R56] Bron, C. and Kerbosch, J. 1973. Algorithm 457: finding all cliques of an undirected graph. Commun. ACM 16, 9 (Sep. 1973), 575-577. http://portal.acm.org/citation.cfm?doid=362342.362367

[R57] Etsuji Tomita, Akira Tanaka, Haruhisa Takahashi, The worst-case time complexity for generating all maximal cliques and computational experiments, Theoretical Computer Science, Volume 363, Issue 1, Computing and Combinatorics, 10th Annual International Conference on Computing and Combinatorics (COCOON 2004), 25 October 2006, Pages 28-42 http://dx.doi.org/10.1016/j.tcs.2006.06.015

[R58] F. Cazals, C. Karande, A note on the problem of reporting maximal cliques, Theoretical Computer Science, Volume 407, Issues 1-3, 6 November 2008, Pages 564-568, http://dx.doi.org/10.1016/j.tcs.2008.05.010

[R44] Depth-first search and linear graph algorithms, R. Tarjan SIAM Journal of Computing 1(2):146-160, (1972).

[R45] On finding the strongly connected components in a directed graph. E. Nuutila and E. Soisalon-Soinen Information Processing Letters 49(1): 9-14, (1994)..

[R46] Depth-first search and linear graph algorithms, R. Tarjan SIAM Journal of Computing 1(2):146-160, (1972).

[R47] On finding the strongly connected components in a directed graph. E. Nuutila and E. Soisalon-Soinen Information Processing Letters 49(1): 9-14, (1994)..

[R59] An O(m) Algorithm for Cores Decomposition of Networks Vladimir Batagelj and Matjaz Zaversnik, 2003 http://arxiv.org/abs/cs.DS/0310049

[R105] Skiena, S. S. The Algorithm Design Manual (Springer-Verlag, 1998). http://www.amazon.com/exec/obidos/ASIN/0387948600/ref=ase_thealgorithmrepo/

[R54] Fleury, "Deux problemes de geometrie de situation", Journal de mathematiques elementaires (1883), 257-261.

[R55] http://en.wikipedia.org/wiki/Eulerian_path

[R95] A. Langville and C. Meyer, "A survey of eigenvector methods of web information retrieval." http://citeseer.ist.psu.edu/713792.html

[R96] 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

[R97] A. Langville and C. Meyer, "A survey of eigenvector methods of web information retrieval." http://citeseer.ist.psu.edu/713792.html

[R98] 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

[R99] A. Langville and C. Meyer, "A survey of eigenvector methods of web information retrieval." http://citeseer.ist.psu.edu/713792.html

[R100] 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

[R86] A. Langville and C. Meyer, "A survey of eigenvector methods of web information retrieval." http://citeseer.ist.psu.edu/713792.html

[R87] 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.

[R88] A. Langville and C. Meyer, "A survey of eigenvector methods of web information retrieval." http://citeseer.ist.psu.edu/713792.html

[R89] 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.

[R90] A. Langville and C. Meyer, "A survey of eigenvector methods of web information retrieval." http://citeseer.ist.psu.edu/713792.html

[R91] 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.

[R93] "Efficient Algorithms for Finding Maximum Matching in Graphs" by Zvi Galil, ACM Computing Surveys, 1986.

[R52] M. E. J. Newman, Mixing patterns in networks, Physical Review E, 67 026126, 2003

[R48] M. E. J. Newman, Mixing patterns in networks, Physical Review E, 67 026126, 2003

[R94] M. E. J. Newman, Mixing patterns in networks Physical Review E, 67 026126, 2003

[R53] M. E. J. Newman, Mixing patterns in networks Physical Review E, 67 026126, 2003

[R78] Batagelj and Brandes, "Efficient generation of large random networks", Phys. Rev. E, 71, 036113, 2005.

[R79] 1. Erdős and A. Rényi, On Random Graphs, Publ. Math. 6, 290 (1959).

[R80] 1. (a) Gilbert, Random Graphs, Ann. Math. Stat., 30, 1141 (1959).

[R74] 1. Erdős and A. Rényi, On Random Graphs, Publ. Math. 6, 290 (1959).

[R75] 1. (a) Gilbert, Random Graphs, Ann. Math. Stat., 30, 1141 (1959).

[R73] Donald E. Knuth, The Art of Computer Programming, Volume 2 / Seminumerical algorithms Third Edition, Addison-Wesley, 1997.

[R76] 1. Erdős and A. Rényi, On Random Graphs, Publ. Math. 6, 290 (1959).

[R77] 1. (a) Gilbert, Random Graphs, Ann. Math. Stat., 30, 1141 (1959).

[R71] 1. Erdős and A. Rényi, On Random Graphs, Publ. Math. 6, 290 (1959).

[R72] 1. (a) Gilbert, Random Graphs, Ann. Math. Stat., 30, 1141 (1959).

[R81] 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

[R85] Duncan J. Watts and Steven H. Strogatz, Collective dynamics of small-world networks, Nature, 393, pp. 440–442, 1998.

[R83] 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

[R84] Jeong Han Kim and Van H. Vu, Generating random regular graphs, Proceedings of the thirty-fifth ACM symposium on Theory of computing, San Diego, CA, USA, pp 213–222, 2003. http://portal.acm.org/citation.cfm?id=780542.780576

[R70] A. L. Barabási and R. Albert "Emergence of scaling in random networks", Science 286, pp 509-512, 1999.

[R82] P. Holme and B. J. Kim, "Growing scale-free networks with tunable clustering", Phys. Rev. E, 65, 026107, 2002.

[R60] M.E.J. Newman, "The structure and function of complex networks", SIAM REVIEW 45-2, pp 167-256, 2003.

[R62] 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)

[R63] Fan Chung and L. Lu, Connected components in random graphs with given expected degree sequences, Ann. Combinatorics, 6, pp. 125-145, 2002.

[R64] G. Chartrand and L. Lesniak, "Graphs and Digraphs", Chapman and Hall/CRC, 1996.

[R65] G. Chartrand and L. Lesniak, "Graphs and Digraphs", Chapman and Hall/CRC, 1996.

[R61] 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

[R66] P. L. Krapivsky and S. Redner, Organization of Growing Random Networks, Phys. Rev. E, 63, 066123, 2001.

[R68] P. L. Krapivsky and S. Redner, Organization of Growing Random Networks, Phys. Rev. E, 63, 066123, 2001.

[R67] P. L. Krapivsky and S. Redner, Network Growth by Copying, Phys. Rev. E, 71, 036118, 2005k.},

[R69] B. Bollob{'a}s, C. Borgs, J. Chayes, and O. Riordan, Directed scale-free graphs, Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms, 132–139, 2003.

[R92] Fan Chung-Graham, Spectral Graph Theory, CBMS Regional Conference Series in Mathematics, Number 92, 1997.

[R102] http://docs.python.org/library/pickle.html

[R106] http://docs.python.org/library/pickle.html

[R103] http://www.algorithmic-solutions.info/leda_guide/graphs/leda_native_graph_fileformat.html

[R101] http://www.algorithmic-solutions.info/leda_guide/graphs/leda_native_graph_fileformat.html

[R104] http://www.yaml.org

[R107] http://www.yaml.org

MODULE INDEX

N

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

W