## CONTENTS

1 Overview ................................................................. 1
   1.1 Who uses NetworkX? ............................................... 1
   1.2 Goals .................................................................. 1
   1.3 The Python programming language .............................. 1
   1.4 Free software ..................................................... 2
   1.5 History .......................................................... 2

2 Introduction .............................................................. 3
   2.1 NetworkX Basics .................................................... 3
   2.2 Nodes and Edges ................................................... 4

3 Graph types .............................................................. 9
   3.1 Which graph class should I use? ................................. 9
   3.2 Basic graph types ................................................. 9

4 Algorithms .................................................................. 127
   4.1 Approximation ...................................................... 127
   4.2 Assortativity ......................................................... 132
   4.3 Bipartite ............................................................ 141
   4.4 Blockmodeling ...................................................... 161
   4.5 Boundary .......................................................... 162
   4.6 Centrality .......................................................... 163
   4.7 Chordal ............................................................. 184
   4.8 Clique ............................................................... 187
   4.9 Clustering .......................................................... 190
   4.10 Communities ....................................................... 193
   4.11 Components ....................................................... 194
   4.12 Connectivity ....................................................... 208
   4.13 Cores ............................................................... 225
   4.14 Cycles ............................................................. 229
   4.15 Directed Acyclic Graphs ......................................... 231
   4.16 Distance Measures ............................................... 234
   4.17 Distance-Regular Graphs ....................................... 236
   4.18 Eulerian ............................................................ 237
   4.19 Flows ............................................................... 239
   4.20 Graphical degree sequence .................................... 263
   4.21 Hierarchy .......................................................... 267
   4.22 Isolates ............................................................ 268
   4.23 Isomorphism ....................................................... 269


<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.24 Link Analysis</td>
<td>282</td>
</tr>
<tr>
<td>4.25 Link Prediction</td>
<td>289</td>
</tr>
<tr>
<td>4.26 Matching</td>
<td>296</td>
</tr>
<tr>
<td>4.27 Maximal independent set</td>
<td>297</td>
</tr>
<tr>
<td>4.28 Minimum Spanning Tree</td>
<td>298</td>
</tr>
<tr>
<td>4.29 Operators</td>
<td>299</td>
</tr>
<tr>
<td>4.30 Rich Club</td>
<td>307</td>
</tr>
<tr>
<td>4.31 Shortest Paths</td>
<td>308</td>
</tr>
<tr>
<td>4.32 Simple Paths</td>
<td>326</td>
</tr>
<tr>
<td>4.33 Swap</td>
<td>327</td>
</tr>
<tr>
<td>4.34 Traversal</td>
<td>328</td>
</tr>
<tr>
<td>4.35 Tree</td>
<td>335</td>
</tr>
<tr>
<td>4.36 Vitality</td>
<td>337</td>
</tr>
<tr>
<td>5 Functions</td>
<td>339</td>
</tr>
<tr>
<td>5.1 Graph</td>
<td>339</td>
</tr>
<tr>
<td>5.2 Nodes</td>
<td>341</td>
</tr>
<tr>
<td>5.3 Edges</td>
<td>342</td>
</tr>
<tr>
<td>5.4 Attributes</td>
<td>343</td>
</tr>
<tr>
<td>5.5 Freezing graph structure</td>
<td>345</td>
</tr>
<tr>
<td>6 Graph generators</td>
<td>347</td>
</tr>
<tr>
<td>6.1 Atlas</td>
<td>347</td>
</tr>
<tr>
<td>6.2 Classic</td>
<td>347</td>
</tr>
<tr>
<td>6.3 Small</td>
<td>352</td>
</tr>
<tr>
<td>6.4 Random Graphs</td>
<td>356</td>
</tr>
<tr>
<td>6.5 Degree Sequence</td>
<td>365</td>
</tr>
<tr>
<td>6.6 Random Clustered</td>
<td>371</td>
</tr>
<tr>
<td>6.7 Directed</td>
<td>372</td>
</tr>
<tr>
<td>6.8 Geometric</td>
<td>375</td>
</tr>
<tr>
<td>6.9 Hybrid</td>
<td>379</td>
</tr>
<tr>
<td>6.10 Bipartite</td>
<td>379</td>
</tr>
<tr>
<td>6.11 Line Graph</td>
<td>383</td>
</tr>
<tr>
<td>6.12 Ego Graph</td>
<td>385</td>
</tr>
<tr>
<td>6.13 Stochastic</td>
<td>386</td>
</tr>
<tr>
<td>6.14 Intersection</td>
<td>386</td>
</tr>
<tr>
<td>6.15 Social Networks</td>
<td>388</td>
</tr>
<tr>
<td>7 Linear algebra</td>
<td>389</td>
</tr>
<tr>
<td>7.1 Graph Matrix</td>
<td>389</td>
</tr>
<tr>
<td>7.2 Laplacian Matrix</td>
<td>391</td>
</tr>
<tr>
<td>7.3 Spectrum</td>
<td>393</td>
</tr>
<tr>
<td>7.4 Algebraic Connectivity</td>
<td>394</td>
</tr>
<tr>
<td>7.5 Attribute Matrices</td>
<td>397</td>
</tr>
<tr>
<td>8 Converting to and from other data formats</td>
<td>403</td>
</tr>
<tr>
<td>8.1 To NetworkX Graph</td>
<td>403</td>
</tr>
<tr>
<td>8.2 Dictionaries</td>
<td>404</td>
</tr>
<tr>
<td>8.3 Lists</td>
<td>405</td>
</tr>
<tr>
<td>8.4 Numpy</td>
<td>406</td>
</tr>
<tr>
<td>8.5 Scipy</td>
<td>411</td>
</tr>
<tr>
<td>9 Reading and writing graphs</td>
<td>415</td>
</tr>
<tr>
<td>9.1 Adjacency List</td>
<td>415</td>
</tr>
<tr>
<td>9.2 Multiline Adjacency List</td>
<td>419</td>
</tr>
</tbody>
</table>
NetworkX is a Python language software package for the creation, manipulation, and study of the structure, dynamics, and function of complex networks.

With NetworkX you can load and store networks in standard and nonstandard data formats, generate many types of random and classic networks, analyze network structure, build network models, design new network algorithms, draw networks, and much more.

1.1 Who uses NetworkX?

The potential audience for NetworkX includes mathematicians, physicists, biologists, computer scientists, and social scientists. Good reviews of the state-of-the-art in the science of complex networks are 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 Goals

NetworkX is intended to provide

- tools for the study of the structure and dynamics of social, biological, and infrastructure networks,
- a standard programming interface and graph implementation that is suitable for many applications,
- a rapid development environment for collaborative, multidisciplinary projects,
- an interface to existing numerical algorithms and code written in C, C++, and FORTRAN,
- the ability to painlessly slurp in large nonstandard data sets.

1.3 The Python programming language

Python is a powerful programming language that allows simple and flexible representations of networks, and clear and concise expressions of network algorithms (and other algorithms too). Python has a vibrant and growing ecosystem of packages that NetworkX uses to provide more features such as numerical linear algebra and drawing. In addition Python is also an excellent “glue” language for putting together pieces of software from other languages which allows reuse of legacy code and engineering of high-performance algorithms [Langtangen04].

Equally important, Python is free, well-supported, and a joy to use.
In order to make the most out of NetworkX you will want to know how to write basic programs in Python. Among the many guides to Python, we recommend the documentation at http://www.python.org and the text by Alex Martelli [Martelli03].

1.4 Free software

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

1.5 History

NetworkX was born in May 2002. The original version was designed and written by Aric Hagberg, Dan Schult, and Pieter Swart in 2002 and 2003. The first public release was in April 2005.

Many people have contributed to the success of NetworkX. Some of the contributors are listed in the credits.

1.5.1 What Next

- A Brief Tour
- Installing
- Reference
- Examples
INTRODUCTION

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)

```python
>>> 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

```python
>>> 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 data structures. The graph adjacency structure is implemented as a Python dictionary of dictionaries; the outer dictionary is keyed by nodes to values that are themselves dictionaries keyed by neighboring node to the edge attributes associated with that edge. This “dict-of-dicts” structure allows fast addition, deletion, and lookup of nodes and neighbors in large graphs. The underlying data structure 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 data structure. This design allows for possible replacement of the ‘dicts-of-dicts’-based data structure with an alternative data structure that implements the same methods.

2.1.1 Graphs

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

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

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

2.2 Nodes and Edges

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

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

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

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

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

2.2.1 Graph Creation

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

- Graph generators – standard algorithms to create network topologies.
- Importing data from pre-existing (usually file) sources.
Explicit addition and removal of nodes/edges is the easiest to describe. Each graph object supplies methods to manipulate the graph. For example,

```python
>>> 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:

```python
>>> 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:

```python
>>> 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 Github 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’)

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

>>> G.add_edge('A','B')

>>> G.add_edge('B','C')

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

The data structure gets morphed slightly for each base graph class. For DiGraph two dict-of-dicts-of-dicts structures are provided, one for successors and one for predecessors. For MultiGraph/MultiDiGraph we use a dict-of-dicts-of-
dicts-of-dicts where the third 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.

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

---

1 "It’s dictionaries all the way down."
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?

<table>
<thead>
<tr>
<th>Graph Type</th>
<th>NetworkX Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undirected Simple</td>
<td>Graph</td>
</tr>
<tr>
<td>Directed Simple</td>
<td>DiGraph</td>
</tr>
<tr>
<td>With Self-loops</td>
<td>Graph, DiGraph</td>
</tr>
<tr>
<td>With Parallel edges</td>
<td>MultiGraph, MultiDiGraph</td>
</tr>
</tbody>
</table>

### 3.2 Basic graph types

#### 3.2.1 Graph – Undirected graphs with self loops

**Overview**

**Graph** *(data=None, **attr]*)

Base class for undirected graphs.

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

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

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

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

**Parameters**

- **data** : input graph

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

- **attr** : keyword arguments, optional (default= no attributes)
Attributes to add to graph as key=value pairs.

See Also:
DiGraph, MultiGraph, MultiDiGraph

Examples

Create an empty graph structure (a “null graph”) with no nodes and no edges.
>>> G = nx.Graph()

G can be grown in several ways.

Nodes:
Add one node at a time:
>>> G.add_node(1)

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

>>> G.add_nodes_from([2,3])
>>> G.add_nodes_from(range(100,110))
>>> H=nx.Graph()
>>> H.add_path([0,1,2,3,4,5,6,7,8,9])
>>> G.add_nodes_from(H)

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

>>> G.add_node(H)

Edges:
G can also be grown by adding edges.

Add one edge,
>>> G.add_edge(1, 2)

a list of edges,
>>> G.add_edges_from([(1,2),(1,3)])

or a collection of edges,
>>> G.add_edges_from(H.edges())

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

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

>>> G = nx.Graph(day="Friday")
>>> G.graph
{’day’: ’Friday’}
Add node attributes using `add_node()`, `add_nodes_from()` or `G.node`

```python
>>> G.add_node(1, time='5pm')
>>> G.add_nodes_from([3], time='2pm')
>>> G.node[1]['time'] = '5pm'
>>> G.node[1]['room'] = 714
>>> del G.node[1]['room']  # remove attribute
>>> G.nodes(data=True)
[(1, {'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`

```python
>>> 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.

```python
>>> 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
```

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

```python
>>> for n,nbrsdic in G.adjacency_iter():
...     for nbr,eattr in nbrsdic.items():
...         if 'weight' in eattr:
...             (n,nbr,eattr['weight'])
(1, 2, 4)
(2, 1, 4)
(2, 3, 8)
(3, 2, 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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Graph.__init__(data)</code></td>
<td>Initialize a graph with edges, name, graph attributes.</td>
</tr>
<tr>
<td><code>Graph.add_node(n, attr_dict)</code></td>
<td>Add a single node n and update node attributes.</td>
</tr>
</tbody>
</table>

3.2. Basic graph types 11
Table 3.1 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Graph.add_nodes_from(nodes, **attr)</code></td>
<td>Add multiple nodes.</td>
</tr>
<tr>
<td><code>Graph.remove_node(n)</code></td>
<td>Remove node n.</td>
</tr>
<tr>
<td><code>Graph.remove_nodes_from(nodes)</code></td>
<td>Remove multiple nodes.</td>
</tr>
<tr>
<td><code>Graph.add_edge(u, v[, attr_dict])</code></td>
<td>Add an edge between u and v.</td>
</tr>
<tr>
<td><code>Graph.add_edges_from(ebunch[, attr_dict])</code></td>
<td>Add all the edges in ebunch.</td>
</tr>
<tr>
<td><code>Graph.add_weighted_edges_from(ebunch[, weight])</code></td>
<td>Add all the edges in ebunch as weighted edges with specified weights.</td>
</tr>
<tr>
<td><code>Graph.remove_edge(u, v)</code></td>
<td>Remove the edge between u and v.</td>
</tr>
<tr>
<td><code>Graph.remove_edges_from(ebunch)</code></td>
<td>Remove all edges specified in ebunch.</td>
</tr>
<tr>
<td><code>Graph.add_star(nodes, **attr)</code></td>
<td>Add a star.</td>
</tr>
<tr>
<td><code>Graph.add_path(nodes, **attr)</code></td>
<td>Add a path.</td>
</tr>
<tr>
<td><code>Graph.add_cycle(nodes, **attr)</code></td>
<td>Add a cycle.</td>
</tr>
<tr>
<td><code>Graph.clear()</code></td>
<td>Remove all nodes and edges from the graph.</td>
</tr>
</tbody>
</table>

**__init__**

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

Initialize a graph with edges, name, graph attributes.

**Parameters**

- **data**: input graph

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

- **name**: string, optional (default='')

  An optional name for the graph.

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

  Attributes to add to graph as key=value pairs.

**See Also**

convert

**Examples**

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

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

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

**add_node**

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

Add a single node n and update node attributes.
Parameters  
n : node

A node can be any hashable Python object except None.

attr_dict : dictionary, optional (default= no attributes)

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

attr : keyword arguments, optional

Set or change attributes using key=value.

See Also:

add_nodes_from

Notes

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

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

Examples

```python
>>> 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:

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

add_nodes_from

Graph.add_nodes_from(nodes, **attr)

Add multiple nodes.

Parameters  nodes : iterable container

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

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

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

See Also:

add_node
Examples

```python
>>> 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.

```python
>>> 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.

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

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

remove_node

```python
Graph.remove_node(n)
```
Remove node n.

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

Parameters  n : node
A node in the graph

Raises  NetworkXError
  If n is not in the graph.

See Also:
remove_nodes_from

Examples

```python
>>> 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()
[]
```

remove_nodes_from

```python
Graph.remove_nodes_from(nodes)
```
Remove multiple nodes.
**Parameters**  
**nodes** : iterable container

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

**See Also:**
remove_node

**Examples**

```python
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()
[]
```

**add_edge**

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

Add an edge between u and v.

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

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

**Parameters**  
**u, v** : nodes

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

**attr_dict** : dictionary, optional (default= no attributes)

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

**attr** : keyword arguments, optional

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

**See Also:**
add_edges_from add a collection of edges

**Notes**

Adding an edge that already exists updates the edge data.

Many NetworkX algorithms designed for weighted graphs use as the edge weight a numerical value assigned to a keyword which by default is '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)

add_edges_from

Graph.add_edges_from(ebunch, attr_dict=None, **attr)
Add all the edges in ebunch.

Parameters  ebunch  : container of edges

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

attr_dict  : dictionary, optional (default= no attributes)

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

attr  : keyword arguments, optional

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

See Also:

add_edge  add a single edge

add_weighted_edges_from  convenient way to add weighted edges

Notes

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

Edge attributes specified in edges as a tuple take precedence over attributes specified generally.

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')
add_weighted_edges_from

NetworkX.add_weighted_edges_from(ebunch, weight='weight', **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.
- **weight**: string, optional (default='weight')
  The attribute name for the edge weights to be added.
- **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 for Graph/DiGraph simply updates the edge data. For MultiGraph/MultiDiGraph,
duplicate edges are stored.

Examples

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

remove_edge

NetworkX.remove_edges_from(u, v)
Remove the edge between nodes 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

**remove_edges_from**

Graph.remove_edges_from(ebunch)
Remove all edges specified in ebunch.

**Parameters**

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

  Each edge given in the list or container will be removed from the graph. The edges can be:
  - 2-tuples (u,v) edge between u and v.
  - 3-tuples (u,v,k) where k is ignored.

**See Also**

remove_edge remove a single edge

**Notes**

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

**Examples**

```python
>>> 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)
```

**add_star**

Graph.add_star(nodes, **attr)
Add a star.

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

**Parameters**

- **nodes**: iterable container

  A container of nodes.

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

  Attributes to add to every edge in star.

**See Also**

add_path, add_cycle
Examples

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

**add_path**

`Graph.add_path(nodes, **attr)`
Add a path.

**Parameters**

- **nodes**: iterable container
  A container of nodes. A path will be constructed from the nodes (in order) and added to the graph.

- **attr**: keyword arguments, optional (default= no attributes)
  Attributes to add to every edge in path.

**See Also:**
add_star, add_cycle

**Examples**

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

**add_cycle**

`Graph.add_cycle(nodes, **attr)`
Add a cycle.

**Parameters**

- **nodes**: iterable container
  A container of nodes. A cycle will be constructed from the nodes (in order) and added to the graph.

- **attr**: keyword arguments, optional (default= no attributes)
  Attributes to add to every edge in cycle.

**See Also:**
add_path, add_star

**Examples**

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

`Graph.clear()`  
Remove all nodes and edges from the graph.  
This also removes the name, and all graph, node, and edge attributes.

Examples

```python
>>> 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

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

nodes

`Graph.nodes(data=False)`  
Return a list of the nodes in the graph.

**Parameters**  
data : boolean, optional (default=False)  
If False return a list of nodes. If True return a two-tuple of node and node data dictionary

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

Examples

```python
>>> 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')
```
nodes_iter

Graph.nodes_iter(data=False)

Return an iterator over the nodes.

Parameters  

  data : boolean, optional (default=False)

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

Returns  

  niter : iterator

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

Notes

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

Examples

Examples

Examples
**edges**

NetworkX Reference, Release 1.9

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

Return a list of edges.

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

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

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

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

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

**Returns** edge_list : list of edge tuples

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

See Also:

**edges_iter** return an iterator over the edges

**Notes**

Nodes in nbunch that are not in the graph will be (quietly) ignored. For directed graphs this returns the out-edges.

**Examples**

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

**edges_iter**

NetworkX Reference, Release 1.9

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

Return an iterator over the edges.

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

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

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

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

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

**Returns** edge_iter : iterator

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

edges  return a list of edges

Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored. For directed graphs this returns the out-edges.

Examples

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

get_edge_data

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

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

Parameters  u,v : nodes

default: any Python object (default=None)

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

Returns  edge_dict : dictionary

The edge attribute dictionary.

Notes

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

```python
>>> 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.

```python
>>> 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 {}
()  # tuple form
{}  # edge not in graph, return 0

neighbors

Graph.neighbors(n)

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

Parameters n : node

A node in the graph

Returns nlist : list

A list of nodes that are adjacent to n.

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]

neighbors_iter

Graph.neighbors_iter(n)

Return an iterator over all neighbors of node n.
Notes

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

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

Examples

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

`__getitem__`

```
Graph.__getitem__(n)
Return a dict of neighbors of node n. Use the expression ‘G[n]’.
```

Parameters:
- `n` : node

A node in the graph.

Returns:
- `adj_dict` : dictionary
  The adjacency dictionary for nodes connected to n.

Notes

G[n] is similar to G.neighbors(n) but the internal data dictionary is returned instead of a list.
Assigning G[n] will corrupt the internal graph data structure. Use G[n] for reading data only.

Examples

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

`adjacency_list`

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

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

Returns:
- `adj_list` : lists of lists
  The adjacency structure of the graph as a list of lists.
See Also:

adjacency_iter

Examples

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

adjacency_iter

Graph.adjacency_iter()
Return an iterator of (node, adjacency dict) tuples for all nodes.
This is the fastest way to look at every edge. For directed graphs, only outgoing adjacencies are included.

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

See Also:

adjacency_list

Examples

```python
>>> 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: {}})]
```

nbunch_iter

Graph.nbunch_iter(nbunch=None)
Return an iterator of nodes contained in nbunch that are also in the graph.
The nodes in nbunch are checked for membership in the graph and if not are silently ignored.

Parameters  nbunch : iterable container, optional (default=all nodes)
A container of nodes. The container will be iterated through once.

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

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

See Also:

Graph.__iter__
Notes

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

To test whether nbunch is a single node, one can use “if nbunch in self:”, even after processing with this routine. If nbunch is not a node or a (possibly empty) sequence/iterator or None, a NetworkXError is raised. Also, if any object in nbunch is not hashable, a NetworkXError is raised.

Information about graph structure

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

has_node

`Graph.has_node(n)`

Return True if the graph contains the node n.

Parameters

- `n`: node

Examples

```python
>>> 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

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

__contains__

`Graph.__contains__(n)`

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

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

**has_edge**

`Graph.has_edge(u, v)`

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

**Parameters**

- `u, v`: nodes

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

**Returns**

- `edge_ind`: bool

  True if edge is in the graph, False otherwise.

**Examples**

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

```python
>>> 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:

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

**order**

`Graph.order()`

Return the number of nodes in the graph.

**Returns**

- `nnodes`: int

  The number of nodes in the graph.

**See Also**

- `number_of_nodes.__len__`
number_of_nodes

Graph\texttt{.number_of_nodes()}  
Return the number of nodes in the graph.

\textbf{Returns} \hspace{1em} \texttt{nnodes} : int  
The number of nodes in the graph.

See Also:
\texttt{order, \_\_len\_}

Examples

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

\_\_len\_

Graph\texttt{\_\_len\_}()  
Return the number of nodes. Use the expression \texttt{`len(G)`}.

\textbf{Returns} \hspace{1em} \texttt{nnodes} : int  
The number of nodes in the graph.

Examples

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

degree

Graph\texttt{.degree}(\texttt{nbunch}=None, \texttt{weight}=None)  
Return the degree of a node or nodes.

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

\textbf{Parameters} \hspace{1em} \texttt{nbunch} : iterable container, optional (default=all nodes)  
A container of nodes. The container will be iterated through once.

\texttt{weight} : string or None, optional (default=None)  
The edge attribute that holds the numerical value used as a weight. If None, then each edge has weight 1. The degree is the sum of the edge weights adjacent to the node.

\textbf{Returns} \hspace{1em} \texttt{nd} : dictionary, or number  
A dictionary with nodes as keys and degree as values or a number if a single node is specified.
Examples

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

degree_iter

Graph.degree_iter(nbunch=None, weight=None)
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.
- **weight**: string or None, optional (default=None)
  The edge attribute that holds the numerical value used as a weight. If None, then each edge has weight 1. The degree is the sum of the edge weights adjacent to the node.

Returns **nd_iter**: an iterator
The iterator returns two-tuples of (node, degree).

See Also:
degree

Examples

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

size

Graph.size(weight=None)
Return the number of edges.

Parameters:
- **weight**: string or None, optional (default=None)
  The edge attribute that holds the numerical value used as a weight. If None, then each edge has weight 1.

Returns **nedges**: int
The number of edges or sum of edge weights 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(weight='weight')
6.0
```

number_of_edges

Graph.number_of_edges(u=None, v=None)
Return the number of edges between two nodes.

Parameters  

- `u`, `v`: nodes, optional (default=all edges)
  
  If `u` and `v` are specified, return the number of edges between `u` and `v`. Otherwise return the total number of all edges.

Returns  

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

See Also:

size

Examples

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

nodes_with_selfloops

Graph.nodes_with_selfloops()
Return a list of nodes with self loops.

A node with a self loop has an edge with both ends adjacent to that node.
Returns node list: list
A list of nodes with self loops.

See Also:
selfloop_edges, number_of_selfloops

Examples

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

selfloop_edges

Graph.selfloop_edges(data=False)
Return a list of selfloop edges.
A selfloop edge has the same node at both ends.

Parameters data: bool, optional (default=False)
Return selfloop edges as two tuples (u,v) (data=False) or three-tuples (u,v,data) (data=True)

Returns edgelist: list of edge tuples
A list of all selfloop edges.

See Also:
nodes_with_selfloops, number_of_selfloops

Examples

```python
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, {})]
```

number_of_selfloops

Graph.number_of_selfloops()
Return the number of selfloop edges.
A selfloop edge has the same node at both ends.

Returns nloops: int
The number of selfloops.
See Also:

```
nodes_with_selfloops, 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

```
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph.copy()</td>
<td>Return a copy of the graph.</td>
</tr>
<tr>
<td>Graph.to_undirected()</td>
<td>Return an undirected copy of the graph.</td>
</tr>
<tr>
<td>Graph.to_directed()</td>
<td>Return a directed representation of the graph.</td>
</tr>
<tr>
<td>Graph.subgraph(nbunch)</td>
<td>Return the subgraph induced on nodes in nbunch.</td>
</tr>
</tbody>
</table>
```

copy

```
Graph.copy()

Return a copy of the graph.

Returns  
G : Graph

A copy of the graph.
```

See Also:

```
to_directed  return a directed copy of the graph.
```

Notes

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

Examples

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

to_undirected

```
Graph.to_undirected()

Return an undirected copy of the graph.

Returns  
G : Graph/MultiGraph

A deepcopy of the graph.
```
See Also:

copy, add_edge, add_edges_from

Notes

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

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

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

Examples

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

te_directed

Graph.to_directed()

Return a directed representation of the graph.

Returns  G : DiGraph

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

Notes

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

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

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

Examples

```python
>>> 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

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

**subgraph**

Graph.subgraph(nbunch)

Return the subgraph induced on nodes in nbunch.

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

**Parameters**

- **nbunch**: list, iterable
  - A container of nodes which will be iterated through once.

**Returns**

- **G**: Graph
  - A subgraph of the graph with the same edge attributes.

**Notes**

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

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

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

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

**Examples**

```python
>>> 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,**attr)

Base class for directed graphs.

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

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

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

Edges are represented as links between nodes with optional key/value attributes.
**Parameters**  
*data* : input graph

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

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

Attributes to add to graph as key=value pairs.

**See Also:**

*Graph, MultiGraph, MultiDiGraph*

**Examples**

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

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

G can be grown in several ways.

**Nodes:**

Add one node at a time:

```python
>>> 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).

```python
>>> 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.

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

**Edges:**

G can also be grown by adding edges.

Add one edge,

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

a list of edges,

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

or a collection of edges,

```python
>>> 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.

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

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

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

```python
>>> 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.

```python
>>> 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
```

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

```python
>>> 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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DiGraph.<strong>init</strong>(data)</td>
<td>Initialize a graph with edges, name, graph attributes.</td>
</tr>
<tr>
<td>DiGraph.add_node(n, attr_dict)</td>
<td>Add a single node n and update node attributes.</td>
</tr>
<tr>
<td>DiGraph.add_nodes_from(nodes, **attr)</td>
<td>Add multiple nodes.</td>
</tr>
<tr>
<td>DiGraph.remove_node(n)</td>
<td>Remove node n.</td>
</tr>
<tr>
<td>DiGraph.remove_nodes_from(nbunch)</td>
<td>Remove multiple nodes.</td>
</tr>
<tr>
<td>DiGraph.add_edge(u, v, attr_dict)</td>
<td>Add an edge between u and v.</td>
</tr>
<tr>
<td>DiGraph.add_edges_from(ebunch, attr_dict)</td>
<td>Add all the edges in ebunch.</td>
</tr>
<tr>
<td>DiGraph.add_weighted_edges_from(ebunch, weight)</td>
<td>Add all the edges in ebunch as weighted edges with specified weights.</td>
</tr>
<tr>
<td>DiGraph.remove_edge(u, v)</td>
<td>Remove the edge between u and v.</td>
</tr>
<tr>
<td>DiGraph.remove_edges_from(ebunch)</td>
<td>Remove all edges specified in ebunch.</td>
</tr>
<tr>
<td>DiGraph.add_star(nodes, **attr)</td>
<td>Add a star.</td>
</tr>
<tr>
<td>DiGraph.add_path(nodes, **attr)</td>
<td>Add a path.</td>
</tr>
<tr>
<td>DiGraph.add_cycle(nodes, **attr)</td>
<td>Add a cycle.</td>
</tr>
<tr>
<td>DiGraph.clear()</td>
<td>Remove all nodes and edges from the graph.</td>
</tr>
</tbody>
</table>

__init__

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

Initialize a graph with edges, name, graph attributes.

**Parameters**

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

- **name** : string, optional (default='')

  An optional name for the graph.

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

  Attributes to add to graph as key=value pairs.

**See Also**

convert

**Examples**

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

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

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

DiGraph\texttt{.add\_node}(n, attr\_dict=None, **attr)

Add a single node \textit{n} and update node attributes.

**Parameters** \textit{n} : node

A node can be any hashable Python object except None.

\textbf{attr\_dict} : dictionary, optional (default= no attributes)

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

\textbf{attr} : keyword arguments, optional

Set or change attributes using key=value.

See Also:

\texttt{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

\begin{verbatim}
>>> 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))
\end{verbatim}

add\_nodes\_from

DiGraph\texttt{.add\_nodes\_from}(nodes, **attr)

Add multiple nodes.

**Parameters** \textbf{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.

\textbf{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,{'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

remove_node

DiGraph.remove_node(n)

Remove node n.

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

Parameters
n : node
  A node in the graph

Raises
NetworkXError
  If n is not in the graph.

See Also:
remove_nodes_from

Examples

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

remove_nodes_from

DiGraph.remove_nodes_from(nbunch)
Remove multiple nodes.

Parameters

\texttt{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()
[]

add_edge

DiGraph.add_edge(u, v, attr_dict=None, **attr)
Add an edge between u and v.

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

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

Parameters

\texttt{u,v} : nodes

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

\texttt{attr_dict} : dictionary, optional (default= no attributes)

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

\texttt{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.

Many NetworkX algorithms designed for weighted graphs use as the edge weight a numerical value assigned to a keyword which by default is ‘weight’.

Examples

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

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

Associate data to edges using keywords:

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

add_edges_from

DiGraph.add_edges_from(ebunch, attr_dict=None, **attr)
Add all the edges in ebunch.

Parameters  
ebunch : container of edges

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

attr_dict : dictionary, optional (default= no attributes)

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

attr : keyword arguments, optional

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

See Also:

add_edge add a single edge
add_weighted_edges_from convenient way to add weighted edges

Notes

Adding the same edge twice has no effect but any edge data will be updated when each duplicate edge is added. Edge attributes specified in edges as a tuple take precedence over attributes specified generally.
Examples

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

Associate data to edges

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

`add_weighted_edges_from`

`DiGraph.add_weighted_edges_from(ebunch, weight='weight', **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.
- `weight` : string, optional (default= 'weight')
  The attribute name for the edge weights to be added.
- `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 for Graph/DiGraph simply updates the edge data. For MultiGraph/MultiDiGraph, duplicate edges are stored.

Examples

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

`remove_edge`

`DiGraph.remove_edge(u, v)`
Remove the edge between u and v.

Parameters
- `u,v` : nodes
  Remove the edge between nodes u and v.

Raises
- `NetworkXError`
If there is not an edge between \( u \) and \( v \).

**See Also:**

`remove_edges_from` remove a collection of edges

**Examples**

```python
>>> 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
```

### remove_edges_from

`DiGraph.remove_edges_from(ebunch)`

Remove all edges specified in \( \text{ebunch} \).

**Parameters**

- **\( \text{ebunch} \):** 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 \( \text{ebunch} \) is not in the graph.

**Examples**

```python
>>> 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)
```

### add_star

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

Add a star.

The first node in \( \text{nodes} \) is the middle of the star. It is connected to all other nodes.

**Parameters**

- **\( \text{nodes} \):** iterable container
A container of nodes.

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

Attributes to add to every edge in star.

**See Also:**

*add_path*, *add_cycle*

**Examples**

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

### add_path

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

Add a path.

**Parameters**

- **nodes**: iterable container
  
  A container of nodes. A path will be constructed from the nodes (in order) and added to the graph.

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

  Attributes to add to every edge in path.

**See Also:**

*add_star*, *add_cycle*

**Examples**

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

### add_cycle

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

Add a cycle.

**Parameters**

- **nodes**: iterable container

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

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

  Attributes to add to every edge in cycle.

**See Also:**

*add_path*, *add_star*
Examples

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

clear

DiGraph.clear()

Remove all nodes and edges from the graph.
This also removes the name, and all graph, node, and edge attributes.

Examples

```python
>>> 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

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

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

Parameters data : boolean, optional (default=False)
If False return a list of nodes. If True return a two-tuple of node and node data dictionary

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

Examples

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

nodes_iter

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

Parameters data : boolean, optional (default=False)
If False the iterator returns nodes. If True return a two-tuple of node and node data dictionary

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

Notes

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

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

Examples

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

>>> [d for n,d in G.nodes_iter(data=True)]
[{}, {}, {}]
NetworkX Reference, Release 1.9

__iter__

DiGraph.__iter__()

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

Returns niter : iterator

An iterator over all nodes in the graph.

Examples

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

edges

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

Return a list of edges.

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

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

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

data : bool, optional (default=False)

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

Returns edge_list: list of edge tuples

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

See Also:

edges_iter return an iterator over the edges

Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored. For directed graphs this returns the out-edges.

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)]
edges_iter

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

Return an iterator over the edges.

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

Parameters

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

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

data : bool, optional (default=False)

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

Returns

edge_iter : iterator

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

See Also:

edges return a list of edges

Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored. For directed graphs this returns the out-edges.

Examples

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

out_edges

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

Return a list of edges.

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

Parameters

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

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

data : bool, optional (default=False)

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

Returns

edge_list: list of edge tuples

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

edges_iter return an iterator over the edges

Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored. For directed graphs this returns the out-edges.

Examples

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

out_edges_iter

DiGraph.out_edges_iter(nbunch=None, data=False) 
Return an iterator over the edges.

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

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

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

data : bool, optional (default=False)

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

Returns edge_iter : iterator

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

See Also:

edges return a list of edges

Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored. For directed graphs this returns the out-edges.

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

in_edges

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

See Also:

g_edges return a list of edges

in_edges_iter

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

Parameters

nbunch : iterable container, optional (default= all nodes)
A container of nodes. The container will be iterated through once.

data : bool, optional (default=False)
If True, return edge attribute dict in 3-tuple (u,v,data).

Returns

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

See Also:

g_edges_iter return an iterator of edges

get_edge_data

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

Parameters

u,v : nodes

default: any Python object (default=None)
Value to return if the edge (u,v) is not found.

Returns

edge_dict : dictionary
The edge attribute dictionary.
Notes

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

```python
>>> 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,

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

Examples

```python
>>> 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
```

neighbors

DiGraph.neighbors(n)

Return a list of successor nodes of n.

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

neighbors_iter

DiGraph.neighbors_iter(n)

Return an iterator over successor nodes of n.

neighbors_iter() and successors_iter() are the same.

__getitem__

DiGraph.__getitem__(n)

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

Parameters  
n : node

A node in the graph.

Returns  
adj_dict : dictionary

The adjacency dictionary for nodes connected to n.
Notes

G[n] is similar to G.neighbors(n) but the internal data dictionary is returned instead of a list.
Assigning G[n] will corrupt the internal graph data structure. Use G[n] for reading data only.

Examples

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

successors

DiGraph.successors(n)
Return a list of successor nodes of n.
neighbors() and successors() are the same function.

successors_iter

DiGraph.successors_iter(n)
Return an iterator over successor nodes of n.
neighbors_iter() and successors_iter() are the same.

predecessors

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

predecessors_iter

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

adjacency_list

DiGraph.adjacency_list()
Return an adjacency list representation of the graph.
The output adjacency list is in the order of G.nodes(). For directed graphs, only outgoing adjacencies are included.

Returns adj_list : lists of lists
The adjacency structure of the graph as a list of lists.

See Also:
adjacency_iter

3.2. Basic graph types 53
Examples

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

adjacency_iter

```python
DiGraph.adjacency_iter()
```

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

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

Returns **adj_iter** : iterator

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

See Also:
adjacency_list

Examples

```python
>>> 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: {}})]
```

nbunch_iter

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

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

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

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

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

Returns **niter** : iterator

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

Raises **NetworkXError**

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

See Also:
Graph.__iter__
Notes

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

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

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

Information about graph structure

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DiGraph.has_node(n)</td>
<td>Return True if the graph contains the node n.</td>
</tr>
<tr>
<td>DiGraph.<strong>contains</strong>(n)</td>
<td>Return True if n is a node. False otherwise. Use the expression</td>
</tr>
<tr>
<td>DiGraph.has_edge(u, v)</td>
<td>Return True if the edge (u,v) is in the graph.</td>
</tr>
<tr>
<td>DiGraph.order()</td>
<td>Return the number of nodes in the graph.</td>
</tr>
<tr>
<td>DiGraph.number_of_nodes()</td>
<td>Return the number of nodes in the graph.</td>
</tr>
<tr>
<td>DiGraph.<strong>len</strong>()</td>
<td>Return the number of nodes.</td>
</tr>
<tr>
<td>DiGraph.degree(nbunch, weight)</td>
<td>Return the degree of a node or nodes.</td>
</tr>
<tr>
<td>DiGraph.in_degree([nbunch, weight])</td>
<td>Return the in-degree of a node or nodes.</td>
</tr>
<tr>
<td>DiGraph.out_degree([nbunch, weight])</td>
<td>Return the out-degree of a node or nodes.</td>
</tr>
<tr>
<td>DiGraph.in_degree_iter([nbunch, weight])</td>
<td>Return an iterator for (node, degree).</td>
</tr>
<tr>
<td>DiGraph.out_degree_iter([nbunch, weight])</td>
<td>Return an iterator for (node, in-degree).</td>
</tr>
<tr>
<td>DiGraph.size([weight])</td>
<td>Return the number of edges.</td>
</tr>
<tr>
<td>DiGraph.number_of_edges([u, v])</td>
<td>Return the number of edges between two nodes.</td>
</tr>
<tr>
<td>DiGraph.nodes_with_selfloops()</td>
<td>Return a list of nodes with self loops.</td>
</tr>
<tr>
<td>DiGraph.selfloop_edges([data])</td>
<td>Return a list of selfloop edges.</td>
</tr>
<tr>
<td>DiGraph.number_of_selfloops()</td>
<td>Return the number of selfloop edges.</td>
</tr>
</tbody>
</table>

has_node

DiGraph.has_node(n)

Return True if the graph contains the node n.

Parameters n : node

Examples

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

It is more readable and simpler to use

>>> 0 in G
True
**__contains__**

`DiGraph.__contains__(n)`  
Return True if n is a node, False otherwise. Use the expression `n in G`.

**Examples**

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

**has_edge**

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

**Parameters**  
`u,v`: nodes  
Nodes can be, for example, strings or numbers. Nodes must be hashable (and not None) Python objects.

**Returns**  
`edge_ind`: bool  
True if edge is in the graph, False otherwise.

**Examples**

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

```python
g = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
g.add_path([0,1,2,3])
g.has_edge(0,1) # using two nodes
```

```python
e = (0,1)
g.has_edge(*e) # e is a 2-tuple (u,v)
```

```python
e = (0,1,{'weight':7})
g.has_edge(*e[:2]) # e is a 3-tuple (u,v, data_dictionary)
```

The following syntax are all equivalent:

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

**order**

`DiGraph.order()`  
Return the number of nodes in the graph.

**Returns**  
`nnodes`: int
The number of nodes in the graph.

See Also:

* `number_of_nodes`
* `__len__`

**number_of_nodes**

`DiGraph.number_of_nodes()`

Return the number of nodes in the graph.

Returns:

- `nnodes`: int
  
  The number of nodes in the graph.

See Also:

* `order`
* `__len__`

**Examples**

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

**__len__**

`DiGraph.__len__()`

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

Returns:

- `nnodes`: int
  
  The number of nodes in the graph.

**Examples**

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

degree

`DiGraph.degree(nbunch=None, weight=None)`

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.

- `weight`: string or None, optional (default=None)

  A node attribute used as a weight. If None, then each edge has weight 1.

**Examples**

```python
>>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.degree(0)
4
```
The edge attribute that holds the numerical value used as a weight. If None, then each
effect has weight 1. The degree is the sum of the 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**

```python
>>> 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]
```

**degree_iter**

DiGraph.**degree_iter**(nbunch=None, weight=None)

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.

**weight** : string or None, optional (default=None)

The edge attribute that holds the numerical value used as a weight. If None, then each
effect has weight 1. The degree is the sum of the 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**

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

**in_degree**

DiGraph.**in_degree**(nbunch=None, weight=None)

Return the in-degree of a node or nodes.

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

weight : string or None, optional (default=None)
    The edge attribute that holds the numerical value used as a weight. If None, then each
    edge has weight 1. The degree is the sum of the 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

```python
>>> 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]
```

in_degree_iter

DiGraph.in_degree_iter (nbunch=None, weight=None)
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.

weight : string or None, optional (default=None)
    The edge attribute that holds the numerical value used as a weight. If None, then each
    edge has weight 1. The degree is the sum of the 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)]

out_degree

DiGraph.out_degree (nbunch=None, weight=None)
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.

- **weight**: string or None, optional (default=None)
  The edge attribute that holds the numerical value used as a weight. If None, then each edge has weight 1. The degree is the sum of the 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]

out_degree_iter

DiGraph.out_degree_iter (nbunch=None, weight=None)
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.

- **weight**: string or None, optional (default=None)
  The edge attribute that holds the numerical value used as a weight. If None, then each edge has weight 1. The degree is the sum of the 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

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

size

DiGraph.size(weight=None)

Return the number of edges.

Parameters

- **weight**: string or None, optional (default=None)
The edge attribute that holds the numerical value used as a weight. If None, then each edge has weight 1.

Returns

- **nedges**: int
The number of edges or sum of edge weights in the graph.

See Also:

number_of_edges

Examples

```python
>>> 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(weight='weight')
6.0
```

number_of_edges

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

Return the number of edges between two nodes.

Parameters

- **u,v**: nodes, optional (default=all edges)
If u and v are specified, return the number of edges between u and v. Otherwise return the total number of all edges.
Returns \texttt{nedges} : \texttt{int}

The number of edges in the graph. If nodes \texttt{u} and \texttt{v} are specified return the number of edges between those nodes.

\textbf{See Also:}

\texttt{size}

\textbf{Examples}

```python
>>> 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
```

\texttt{nodes\_with\_selfloops}

\texttt{DiGraph.nodes\_with\_selfloops}()

Return a list of nodes with self loops.

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

Returns \texttt{nodelist} : \texttt{list}

A list of nodes with self loops.

\textbf{See Also:}

\texttt{selfloop\_edges, number\_of\_selfloops}

\textbf{Examples}

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

\texttt{selfloop\_edges}

\texttt{DiGraph.selfloop\_edges}\texttt{(data=False)}

Return a list of selfloop edges.

A selfloop edge has the same node at both ends.

\textbf{Parameters} \texttt{data} : bool, optional (default=False)

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

Returns \texttt{edgelist} : list of edge tuples
A list of all selfloop edges.

See Also:

nodes_with_selfloops, number_of_selfloops

Examples

```python
>>> 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, {})]
```

number_of_selfloops

DiGraph.number_of_selfloops()

Return the number of selfloop edges. A selfloop edge has the same node at both ends.

Returns nloops : int

The number of selfloops.

See Also:

nodes_with_selfloops, selfloop_edges

Examples

```python
>>> 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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DiGraph.copy()</td>
<td>Return a copy of the graph.</td>
</tr>
<tr>
<td>DiGraph.to_undirected() ([reciprocal])</td>
<td>Return an undirected representation of the digraph.</td>
</tr>
<tr>
<td>DiGraph.to_directed()</td>
<td>Return a directed copy of the graph.</td>
</tr>
<tr>
<td>DiGraph.subgraph(nbunch)</td>
<td>Return the subgraph induced on nodes in nbunch.</td>
</tr>
<tr>
<td>DiGraph.reverse([copy])</td>
<td>Return the reverse of the graph.</td>
</tr>
</tbody>
</table>

copy

DiGraph.copy()

Return a copy of the graph.

Returns G : Graph

3.2. Basic graph types 63
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

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

**to_undirected**

DiGraph.to_undirected(*reciprocal=\texttt{False}*)

Return an undirected representation of the digraph.

Parameters **reciprocal** : bool (optional)

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

Returns **G** : Graph

An undirected graph with the same name and nodes and with edge \( (u,v,\text{data}) \) if either \( (u,v,\text{data}) \) or \( (v,u,\text{data}) \) 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 \( \text{G=DiGraph(D)} \) 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.

**to_directed**

DiGraph.to_directed()

Return a directed copy of the graph.

Returns **G** : DiGraph

A deepcopy of the graph.
Notes

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

This is in contrast to the similar \( D = \text{DiGraph}(G) \) 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

```python
>>> 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

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

subgraph

DiGraph.\texttt{subgraph}(\texttt{nbunch})

Return the subgraph induced on nodes in \texttt{nbunch}.

The induced subgraph of the graph contains the nodes in \texttt{nbunch} and the edges between those nodes.

Parameters  \texttt{nbunch} : list, iterable

A container of nodes which will be iterated through once.

Returns  \texttt{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

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

reverse

```python```

DiGraph.reverse(copy=True)

Return the reverse of the graph.

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

Parameters copy : bool optional (default=True)

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

3.2.3 MultiGraph - Undirected graphs with self loops and parallel edges

Overview

MultiGraph(data=None, **attr)

An undirected graph class that can store multiedges.

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

A MultiGraph holds undirected edges. Self loops are allowed.

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

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

Parameters data : input graph

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

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

Attributes to add to graph as key=value pairs.

See Also:

Graph, DiGraph, MultiDiGraph

Examples

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

```python
>>> G = nx.MultiGraph()
```
G can be grown in several ways.

Nodes:

Add one node at a time:

```python
>>> 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).

```python
>>> 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.

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

Edges:

G can also be grown by adding edges.

Add one edge,

```python
>>> G.add_edge(1, 2)
```
a list of edges,

```python
>>> G.add_edges_from([(1, 2), (1, 3)])
```
or a collection of edges,

```python
>>> 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.

```python
>>> 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.

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

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

```python
>>> G.add_node(1, time='5pm')
>>> G.add_nodes_from([3], time='2pm')
>>> G.node[1]
{'time': '5pm'}
>>> G.node[1]['room'] = 714
>>> del G.node[1]['room'] # remove attribute
```
>>> G.nodes(data=True)
[(1, {'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, nbrdict in G.adjacency_iter():
...   for nbr, keydict in nbrdict.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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MultiGraph.<strong>init</strong>(data)</td>
<td>Initialize a graph with edges, name, graph attributes.</td>
</tr>
<tr>
<td>MultiGraph.add_node(n[, attr_dict])</td>
<td>Add a single node n and update node attributes.</td>
</tr>
<tr>
<td>MultiGraph.add_nodes_from(nodes, **attr)</td>
<td>Add multiple nodes.</td>
</tr>
<tr>
<td>MultiGraph.remove_node(n)</td>
<td>Remove node n.</td>
</tr>
<tr>
<td>MultiGraph.remove_nodes_from(nodes)</td>
<td>Remove multiple nodes.</td>
</tr>
</tbody>
</table>
Table 3.9 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MultiGraph.add_edge(u, v[, key, attr_dict])</td>
<td>Add an edge between u and v.</td>
</tr>
<tr>
<td>MultiGraph.add_edges_from(ebunch[, attr_dict])</td>
<td>Add all the edges in ebunch.</td>
</tr>
<tr>
<td>MultiGraph.add_weighted_edges_from(ebunch[, ...])</td>
<td>Add all the edges in ebunch as weighted edges with specified weights.</td>
</tr>
<tr>
<td>MultiGraph.remove_edge(u, v[, key])</td>
<td>Remove an edge between u and v.</td>
</tr>
<tr>
<td>MultiGraph.remove_edges_from(ebunch)</td>
<td>Remove all edges specified in ebunch.</td>
</tr>
<tr>
<td>MultiGraph.add_star(nodes, **attr)</td>
<td>Add a star.</td>
</tr>
<tr>
<td>MultiGraph.add_path(nodes, **attr)</td>
<td>Add a path.</td>
</tr>
<tr>
<td>MultiGraph.add_cycle(nodes, **attr)</td>
<td>Add a cycle.</td>
</tr>
<tr>
<td>MultiGraph.clear()</td>
<td>Remove all nodes and edges from the graph.</td>
</tr>
</tbody>
</table>

__init__

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

Initialize a graph with edges, name, graph attributes.

Parameters

- data: input graph
  Data to initialize graph. If data=None (default) an empty graph is created. The data can be an edge list, or any NetworkX graph object. If the corresponding optional Python packages are installed the data can also be a NumPy matrix or 2d ndarray, a SciPy sparse matrix, or a PyGraphviz graph.
- name: string, optional (default='')
  An optional name for the graph.
- attr: keyword arguments, optional (default= no attributes)
  Attributes to add to graph as key=value pairs.

See Also:

convert

Examples

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

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

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

add_node

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

Add a single node n and update node attributes.

Parameters

- n: node
  A node can be any hashable Python object except None.
attr_dict : dictionary, optional (default= no attributes)
    Dictionary of node attributes. Key/value pairs will update existing data associated with
    the node.

attr : keyword arguments, optional
    Set or change attributes using key=value.

See Also:
add_nodes_from

Notes
A hashable object is one that can be used as a key in a Python dictionary. This includes strings, numbers, tuples
of strings and numbers, etc.
On many platforms hashable items also include mutables such as NetworkX Graphs, though one should be
careful that the hash doesn’t change on mutables.

Examples

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

add_nodes_from

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

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

attr : keyword arguments, optional (default= no attributes)
    Update attributes for all nodes in nodes. Node attributes specified in nodes as a tuple
    take precedence over attributes specified generally.

See Also:
add_node
Examples

```python
>>> 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.

```python
>>> 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.

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

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

`remove_node`

MultiGraph.remove_node(n)

Remove node n.

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

Parameters

- **n**: node
  
  A node in the graph

 Raises

NetworkXError

If n is not in the graph.

See Also:

- remove_nodes_from

Examples

```python
>>> 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()
[]
```

`remove_nodes_from`

MultiGraph.remove_nodes_from(nodes)

Remove multiple nodes.
Parameters  

**nodes** : iterable container

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

See Also:

```
remove_node
```

Examples

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

### add_edge

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

Add an edge between u and v.

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

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

**Parameters**  

**u, v** : nodes

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

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

Used to distinguish multiedges between a pair of nodes.

**attr_dict** : dictionary, optional (default= no attributes)

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

**attr** : keyword arguments, optional

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

See Also:

```
add_edges_from add a collection of edges
```

Notes

To replace/update edge data, use the optional key argument to identify a unique edge. Otherwise a new edge will be created.
NetworkX algorithms designed for weighted graphs cannot use multigraphs directly because it is not clear how to handle multiedge weights. Convert to Graph using edge attribute ‘weight’ to enable weighted graph algorithms.

**Examples**

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

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

Associate data to edges using keywords:

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

### add_edges_from

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

Add all the edges in ebunch.

**Parameters**

ebunch : container of edges

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

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

attr_dict : dictionary, optional (default= no attributes)

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

attr : keyword arguments, optional

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

**See Also:**

- add_edge add a single edge
- add_weighted_edges_from convenient way to add weighted edges

**Notes**

Adding the same edge twice has no effect but any edge data will be updated when each duplicate edge is added. Edge attributes specified in edges as a tuple take precedence over attributes specified generally.
Examples

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

Associate data to edges

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

**add_weighted_edges_from**

```python
MultiGraph.add_weighted_edges_from(ebunch, weight='weight', **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.

- **weight**: string, optional (default='weight')
  
  The attribute name for the edge weights to be added.

- **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 for Graph/DiGraph simply updates the edge data. For MultiGraph/MultiDiGraph,
duplicate edges are stored.

**Examples**

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

**remove_edge**

```python
MultiGraph.remove_edge(u, v, key=None)
```

Remove an edge between u and v.

**Parameters**

- **u, v**: nodes
  
  Remove an edge between nodes u and v.

- **key**: hashable identifier, optional (default=None)
Used to distinguish multiple edges between a pair of nodes. If None remove a single (arbitrary) edge between u and v.

**Raises** NetworkXError

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

**See Also:**

remove_edges_from remove a collection of edges

**Examples**

```python
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

```python
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

```python
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')
```

**remove_edges_from**

MultiGraph.remove_edges_from(ebunch)

Remove all edges specified in ebunch.

**Parameters**

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

  Each edge given in the list or container will be removed from the graph. The edges can be:
  - 2-tuples (u,v) All edges between u and v are removed.
  - 3-tuples (u,v,key) The edge identified by key is removed.
  - 4-tuples (u,v,key,data) where data 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

```python
>>> 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
```  
```python
>>> 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
[]
```

### add_star

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

Add a star.

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

**Parameters**

- `nodes`: iterable container
  A container of nodes.

- `attr`: keyword arguments, optional (default= no attributes)
  Attributes to add to every edge in star.

**See Also:**

add_path, add_cycle

**Examples**

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

### add_path

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

Add a path.

**Parameters**

- `nodes`: iterable container
  A container of nodes. A path will be constructed from the nodes (in order) and added to the graph.

- `attr`: keyword arguments, optional (default= no attributes)
  Attributes to add to every edge in path.
See Also:

add_star, add_cycle

Examples

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

add_cycle

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

Add a cycle.

Parameters

- **nodes**: iterable container
  
  A container of nodes. A cycle will be constructed from the nodes (in order) and added to the graph.

- **attr**: keyword arguments, optional (default= no attributes)
  
  Attributes to add to every edge in cycle.

See Also:

add_path, add_star

Examples

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

clear

```python
MultiGraph.clear()
```

Remove all nodes and edges from the graph.

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

Examples

```python
>>> 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.

## nodes

### MultiGraph.nodes(data=False)

Return a list of the nodes in the graph.

    Parameters data : boolean, optional (default=False)
        If False return a list of nodes. If True return a two-tuple of node and node data dictionary

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

### Examples

```python
>>> 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, {})]
```

## nodes_iter

### MultiGraph.nodes_iter(data=False)

Return an iterator over the nodes.

    Parameters data : boolean, optional (default=False)
        If False the iterator returns nodes. If True return a two-tuple of node and node data dictionary

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

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

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

Examples

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

```
[{}, {}, {}]
```

`__iter__`

`MultiGraph.__iter__()`

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

**Returns**  
`niter`: iterator

An iterator over all nodes in the graph.

Examples

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

`edges`

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

Return a list of edges.

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

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

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

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

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

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

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

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

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

**See Also:**

`edges_iter` return an iterator over the edges
Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored. For directed graphs this returns the out-edges.

Examples

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

edges_iter

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

Return an iterator over the edges.

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

Parameters

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

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

data : bool, optional (default=False)

If True, return edge attribute dict with each edge.

keys : bool, optional (default=False)

If True, return edge keys with each edge.

Returns

edge_iter : iterator

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

See Also:

edges return a list of edges

Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored. For directed graphs this returns the out-edges.
Examples

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

**get_edge_data**

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

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

**Parameters**

- **u,v**: nodes
- **key**: hashable identifier, optional (default=None)
- **default**: any Python object (default=None)

**Returns**

- **edge_dict**: dictionary
  
The edge attribute dictionary.

**Notes**

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

```python
>>> 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.

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

```python
>>> 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
```

neighbors

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

Parameters
  n : node
    A node in the graph

Returns
  nlist : list
    A list of nodes that are adjacent to n.

Raises
  NetworkXError
    If the node n is not in the graph.

Notes

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

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

Examples

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

neighbors_iter

`MultiGraph.neighbors_iter(n)`
Return an iterator over all neighbors of node n.
Notes

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

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

Examples

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

`__getitem__`

MultiGraph.__getitem__(n)
Return a dict of neighbors of node n. Use the expression ‘G[n]’.

Parameters  
- `n`: node
  A node in the graph.

Returns  
- `adj_dict`: dictionary
  The adjacency dictionary for nodes connected to n.

Notes

G[n] is similar to G.neighbors(n) but the internal data dictionary is returned instead of a list. Assigning G[n] will corrupt the internal graph data structure. Use G[n] for reading data only.

Examples

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

`adjacency_list`

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

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

Returns  
- `adj_list`: lists of lists
  The adjacency structure of the graph as a list of lists.
See Also:

adjacency_iter

Examples

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

**adjacency_iter**

MultiGraph.**adjacency_iter**

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

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

Returns **adj_iter** : iterator

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

See Also:

adjacency_list

Examples

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

**nbunch_iter**

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

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

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

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

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

Returns **niter** : iterator

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

Raises **NetworkXError**

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

See Also:

Graph.__iter__
Notes

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

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

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

Information about graph structure

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

**has_node**

MultiGraph.has_node(n)
Return True if the graph contains the node n.

**Parameters**

- **n**: node

**Examples**

```python
>>> 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

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

**__contains__**

MultiGraph.__contains__(n)
Return True if n is a node, False otherwise. Use the expression ‘n in G’.
Examples

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

**has_edge**

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

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

**Parameters**

- u, v : nodes
  - Nodes can be, for example, strings or numbers.
- key : hashable identifier, optional (default=None)
  - If specified return True only if the edge with key is found.

**Returns**

- edge_ind : bool
  - True if edge is in the graph, False otherwise.

Examples

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

```python
>>> 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:

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

**order**

```
MultiGraph.order()
```

Return the number of nodes in the graph.

**Returns**

- nnodes : int
  - The number of nodes in the graph.
See Also:

`number_of_nodes`, `__len__`

**number_of_nodes**

`MultiGraph.number_of_nodes()`  
Return the number of nodes in the graph.  

**Returns**  

`nnodes`: int  
The number of nodes in the graph.

See Also:

`order`, `__len__`

**Examples**

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

**__len__**

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

**Returns**  

`nnodes`: int  
The number of nodes in the graph.

**Examples**

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

**degree**

`MultiGraph.degree(nbunch=None, weight=None)`  
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.  

`weight`: string or None, optional (default=None)  
The edge attribute that holds the numerical value used as a weight. If None, then each edge has weight 1. The degree is the sum of the 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

```python
>>> 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]
```

\textbf{degree_iter}

\texttt{MultiGraph.degree_iter}(\texttt{nbunch}=None, \texttt{weight}=None)

Return an iterator for (node, degree).

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

\textbf{Parameters} \texttt{nbunch} : iterable container, optional (default=all nodes)

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

\texttt{weight} : string or None, optional (default=None)

The edge attribute that holds the numerical value used as a weight. If None, then each edge has weight 1. The degree is the sum of the edge weights adjacent to the node.

Returns \( nd \texttt{_iter} \) : an iterator

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

\textbf{See Also:}

degree

Examples

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

\textbf{size}

\texttt{MultiGraph.size}(\texttt{weight}=None)

Return the number of edges.

\textbf{Parameters} \texttt{weight} : string or None, optional (default=None)
The edge attribute that holds the numerical value used as a weight. If None, then each edge has weight 1.

Returns nedges : int

The number of edges or sum of edge weights in the graph.

See Also:

number_of_edges

Examples

```python
>>> 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(weight='weight')
6.0
```

**number_of_edges**

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

Return the number of edges between two nodes.

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

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

Returns nedges : int

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

See Also:

size

Examples

```python
>>> 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
```
nodes_with_selfloops

`MultiGraph.nodes_with_selfloops()`

Return a list of nodes with self loops.

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

Returns `nodelist` : list

A list of nodes with self loops.

See Also:

`selfloop_edges`, `number_of_selfloops`

Examples

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

selfloop_edges

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

Return a list of selfloop edges.

A selfloop edge has the same node at both ends.

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

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

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

If True, return edge keys with each edge.

Returns `edgelist` : list of edge tuples

A list of all selfloop edges.

See Also:

`nodes_with_selfloops`, `number_of_selfloops`

Examples

```python
>>> 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, {})]]

number_of_selfloops

MultiGraph.number_of_selfloops()
Return the number of selfloop edges.
A selfloop edge has the same node at both ends.

Returns nloops : int
The number of selfloops.

See Also:

nodes_with_selfloops, 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

copy

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.

Notes

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

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

### to_undirected

`MultiGraph.to_undirected()`  
Return an undirected copy of the graph.  

**Returns**  
$G$ : Graph/MultiGraph  
A deep copy of the graph.

**See Also:**

`copy`, `add_edge`, `add_edges_from`

**Notes**

This returns a “deep copy” 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](http://docs.python.org/library/copy.html).

**Examples**

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

### to_directed

`MultiGraph.to_directed()`  
Return a directed representation of the graph.

**Returns**  
$G$ : MultiDiGraph  
A directed graph with the same name, same nodes, and with each edge $(u,v,\text{data})$ replaced by two directed edges $(u,v,\text{data})$ and $(v,u,\text{data})$.  

3.2. Basic graph types
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

```python
>>> 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

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

subgraph

MultiGraph.subgraph(nbunch)

Return the subgraph induced on nodes in nbunch.

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

Parameters  nbunch : list, iterable

A container of nodes which will be iterated through once.

Returns  G : Graph

A subgraph of the graph with the same edge attributes.

Notes

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

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

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

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

```python
>>> 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, **attr``)

A directed graph class that can store multiedges.

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

A MultiDiGraph holds directed edges. Self loops are allowed.

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

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

**Parameters**

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

- **attr** : keyword arguments, optional (default= no attributes)
  
  Attributes to add to graph as key=value pairs.

**See Also:**

- Graph
- DiGraph
- MultiGraph

Examples

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

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

G can be grown in several ways.

**Nodes:**

Add one node at a time:

```python
>>> 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).

```python
>>> 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.

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

**Edges:**

G can also be grown by adding edges.

Add one edge,

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

a list of edges,

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

or a collection of edges,

```python
>>> 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.

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

```python
>>> 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.

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

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

```python
>>> G.add_node(1, time='5pm')
>>> G.add_nodes_from([3], time='2pm')
```

```python
>>> G.node[1]
{'time': '5pm'}
```

```python
>>> del G.node[1]['room']
# remove attribute
>>> G.nodes(data=True)
[(1, {'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.

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

```python
>>> 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.
### Adding and Removing Nodes and Edges

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

### Added and Removed Edges

```python
>>> 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.

```python
>>> 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)
```

### 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__(data)** Initialize a graph with edges, name, graph attributes.
- **MultiDiGraph.add_node(n[, 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[, key, attr_dict])** Add an edge between u and v.
- **MultiDiGraph.add_edges_from(ebunch[, attr_dict])** Add all the edges in ebunch.
- **MultiDiGraph.add_weighted_edges_from(ebunch)** Add all the edges in ebunch as weighted edges with specified weights.
- **MultiDiGraph.remove_edge(u, v[, key])** Remove an edge between u and v.
- **MultiDiGraph.remove_edges_from(ebunch)** Remove all edges specified in ebunch.
- **MultiDiGraph.add_star(nodes[, **attr])** Add a star.
- **MultiDiGraph.add_path(nodes[, **attr])** Add a path.
- **MultiDiGraph.add_cycle(nodes[, **attr])** Add a cycle.
- **MultiDiGraph.clear()** Remove all nodes and edges from the graph.

### Added and Removed Edges

```python
>>> 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)
```

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

**name**: string, optional (default='')

An optional name for the graph.

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

Attributes to add to graph as key=value pairs.

**See Also:**

convert

**Examples**

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

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

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

### add_node

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

Add a single node n and update node attributes.

**Parameters**

- **n**: node
  
  A node can be any hashable Python object except None.
  
- **attr_dict**: dictionary, optional (default= no attributes)
  
  Dictionary of node attributes. Key/value pairs will update existing data associated with the node.
  
- **attr**: keyword arguments, optional
  
  Set or change attributes using key=value.

**See Also:**

add_nodes_from

**Notes**

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

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

```python
>>> 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:

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

add_nodes_from

MultiDiGraph.add_nodes_from(nodes, **attr)

Add multiple nodes.

Parameters

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

- `attr`: keyword arguments, optional (default= no attributes)
  Update attributes for all nodes in nodes. Node attributes specified in nodes as a tuple take precedence over attributes specified generally.

See Also:

- `add_node`

Examples

```python
>>> 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.

```python
>>> G.add_nodes_from([(1,dict(size=10)), (2,{'color':'blue'})])
>>> G.node[1]['size']
11
```

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

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

3.2. Basic graph types
remove_node

MultiDiGraph.remove_node(n)

Remove node n.

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

Parameters n : node

A node in the graph

Raises NetworkXError

If n is not in the graph.

See Also:

remove_nodes_from

Examples

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

remove_nodes_from

MultiDiGraph.remove_nodes_from(nbunch)

Remove multiple nodes.

Parameters nodes : iterable container

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

See Also:

remove_node

Examples

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

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

Add an edge between u and v.

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

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

Parameters

u, v : nodes

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

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

Used to distinguish multiedges between a pair of nodes.

attr_dict : dictionary, optional (default= no attributes)

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

attr : keyword arguments, optional

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

See Also:

add_edges_from add a collection of edges

Notes

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

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

Examples

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

```python
>>> 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:

```python
>>> 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)
```
add_edges_from

MultiDiGraph.add_edges_from(ebunch, attr_dict=None, **attr)
Add all the edges in ebunch.

Parameters

- **ebunch**: container of edges
  Each edge given in the container will be added to the graph. The edges can be:
  - 2-tuples (u,v) or
  - 3-tuples (u,v,d) for an edge attribute dict d, or
  - 4-tuples (u,v,k,d) for an edge identified by key k

- **attr_dict**: dictionary, optional (default= no attributes)
  Dictionary of edge attributes. Key/value pairs will update existing data associated with each edge.

- **attr**: keyword arguments, optional
  Edge data (or labels or objects) can be assigned using keyword arguments.

See Also:

- add_edge  add a single edge
- add_weighted_edges_from  convenient way to add weighted edges

Notes

Adding the same edge twice has no effect but any edge data will be updated when each duplicate edge is added. Edge attributes specified in edges as a tuple take precedence over attributes specified generally.

Examples

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

Associate data to edges

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

add_weighted_edges_from

MultiDiGraph.add_weighted_edges_from(ebunch, weight='weight', **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.

- **weight**: string, optional (default= ‘weight’)
  Edge data (or labels or objects) can be assigned using keyword arguments.
The attribute name for the edge weights to be added.

**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 for Graph/DiGraph simply updates the edge data. For MultiGraph/MultiDiGraph, duplicate edges are stored.

**Examples**

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

**remove_edge**

`MultiDiGraph.remove_edge(u, v, key=None)`

Remove an edge between u and v.

**Parameters**

- `u, v`: nodes
  - Remove an edge between nodes u and v.

- `key`: hashable identifier, optional (default=None)
  - Used to distinguish multiple edges between a pair of nodes. If None remove a single (arbitrary) edge between u and v.

**Raises**

`NetworkXError`

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

**See Also:**

- `remove_edges_from` remove a collection of edges

**Examples**

```python
>>> 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')

**remove_edges_from**

MultiDiGraph.remove_edges_from(ebunch)
Remove all edges specified in ebunch.

Parameters

ebunch: list or container of edge tuples

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

- 2-tuples (u,v) All edges between u and v are removed.
- 3-tuples (u,v,key) The edge identified by key is removed.
- 4-tuples (u,v,key,data) where data 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.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 Reference, Release 1.9

add_star

MultiDiGraph.add_star(nodes, **attr)
Add a star.
The first node in nodes is the middle of the star. It is connected to all other nodes.

Parameters
nodes : iterable container
A container of nodes.

attr : keyword arguments, optional (default= no attributes)
Attributes to add to every edge in star.

See Also:
add_path, add_cycle

Examples

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

add_path

MultiDiGraph.add_path(nodes, **attr)
Add a path.

Parameters
nodes : iterable container
A container of nodes. A path will be constructed from the nodes (in order) and added
to the graph.

attr : keyword arguments, optional (default= no attributes)
Attributes to add to every edge in path.

See Also:
add_star, add_cycle

Examples

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

add_cycle

MultiDiGraph.add_cycle(nodes, **attr)
Add a cycle.

Parameters
nodes : iterable container

3.2. Basic graph types 105
A container of nodes. A cycle will be constructed from the nodes (in order) and added to the graph.

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

Attributes to add to every edge in cycle.

**See Also:**

add_path, add_star

**Examples**

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

**clear**

MultiDiGraph.clear()

Remove all nodes and edges from the graph.

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

**Examples**

```python
>>> 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**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MultiDiGraph.nodes([data])</td>
<td>Return a list of the nodes in the graph.</td>
</tr>
<tr>
<td>MultiDiGraph.nodes_iter([data])</td>
<td>Return an iterator over the nodes.</td>
</tr>
<tr>
<td>MultiDiGraph.<strong>iter</strong>()</td>
<td>Iterate over the nodes.</td>
</tr>
<tr>
<td>MultiDiGraph.edges([nbunch, data, keys])</td>
<td>Return a list of edges.</td>
</tr>
<tr>
<td>MultiDiGraph.edges_iter([nbunch, data, keys])</td>
<td>Return an iterator over the edges.</td>
</tr>
<tr>
<td>MultiDiGraph.out_edges([nbunch, keys, data])</td>
<td>Return a list of the outgoing edges.</td>
</tr>
<tr>
<td>MultiDiGraph.out_edges_iter([nbunch, data, keys])</td>
<td>Return an iterator over the edges.</td>
</tr>
<tr>
<td>MultiDiGraph.in_edges([nbunch, keys, data])</td>
<td>Return a list of the incoming edges.</td>
</tr>
<tr>
<td>MultiDiGraph.in_edges_iter([nbunch, data, keys])</td>
<td>Return an iterator over the incoming edges.</td>
</tr>
<tr>
<td>MultiDiGraph.get_edge_data(u, v[, key, default])</td>
<td>Return the attribute dictionary associated with edge (u,v).</td>
</tr>
<tr>
<td>MultiDiGraph.neighbors(n)</td>
<td>Return a list of successor nodes of n.</td>
</tr>
<tr>
<td>MultiDiGraph.neighbors_iter(n)</td>
<td>Return an iterator over successor nodes of n.</td>
</tr>
<tr>
<td>MultiDiGraph.<strong>getitem</strong>(n)</td>
<td>Return a dict of neighbors of node n.</td>
</tr>
<tr>
<td>MultiDiGraph.successors(n)</td>
<td>Return a list of successor nodes of n.</td>
</tr>
<tr>
<td>MultiDiGraph.successors_iter(n)</td>
<td>Return an iterator over successor nodes of n.</td>
</tr>
</tbody>
</table>
Table 3.14 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MultiDiGraph.predecessors(n)</td>
<td>Return a list of predecessor nodes of n.</td>
</tr>
<tr>
<td>MultiDiGraph.predecessors_iter(n)</td>
<td>Return an iterator over predecessor nodes of n.</td>
</tr>
<tr>
<td>MultiDiGraph.adjacency_list()</td>
<td>Return an adjacency list representation of the graph.</td>
</tr>
<tr>
<td>MultiDiGraph.adjacency_iter()</td>
<td>Return an iterator of (node, adjacency dict) tuples for all nodes.</td>
</tr>
<tr>
<td>MultiDiGraph.nbunch_iter([nbunch])</td>
<td>Return an iterator of nodes contained in nbunch that are also in the graph.</td>
</tr>
</tbody>
</table>

**nodes**

MultiDiGraph.nodes *(data=False)*

Return a list of the nodes in the graph.

**Parameters**

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

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

**Returns**

- **nlist**: list

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

**Examples**

```python
>>> 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, {})]
```

**nodes_iter**

MultiDiGraph.nodes_iter *(data=False)*

Return an iterator over the nodes.

**Parameters**

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

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

**Returns**

- **niter**: iterator

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

**Notes**

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

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

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

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

__iter__

MultiDiGraph.__iter__() Iterate over the nodes. Use the expression ‘for n in G’.

**Returns**

- **niter**: iterator

  An iterator over all nodes in the graph.

Examples

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

edges

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

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

**Parameters**

- **nbunch**: iterable container, optional (default= all nodes)
  
  A container of nodes. The container will be iterated through once.

- **data**: bool, optional (default=False)
  
  Return two tuples (u,v) (False) or three-tuples (u,v,data) (True).

- **keys**: bool, optional (default=False)
  
  Return two tuples (u,v) (False) or three-tuples (u,v,key) (True).

**Returns**

- **edge_list**: list of edge tuples

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

**See Also:**

- **edges_iter** return an iterator over the edges

**Notes**

Nodes in nbunch that are not in the graph will be (quietly) ignored. For directed graphs this returns the out-edges.
Examples

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

**edges_iter**

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

Return an iterator over the edges.

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

**Parameters**

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

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

data : bool, optional (default=False)

If True, return edge attribute dict with each edge.

keys : bool, optional (default=False)

If True, return edge keys with each edge.

**Returns**

edge_iter : iterator

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

**See Also:**

edges return a list of edges

**Notes**

Nodes in nbunch that are not in the graph will be (quietly) ignored. For directed graphs this returns the out-edges.

**Examples**

```python
>>> G = nx.MultiGraph()
>>> 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]))
```
-networkx Reference, Release 1.9-

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

**out_edges**

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

Return a list of the outgoing edges.

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

**Parameters**

- **nbunch**: iterable container, optional (default=all nodes)
  A container of nodes. The container will be iterated through once.

- **data**: bool, optional (default=False)
  If True, return edge attribute dict with each edge.

- **keys**: bool, optional (default=False)
  If True, return edge keys with each edge.

**Returns**

- **out_edges**: list
  An list of (u,v), (u,v,d) or (u,v,key,d) tuples of edges.

**See Also**

- **in_edges** return a list of incoming edges

**Notes**

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

**out_edges_iter**

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

Return an iterator over the edges.

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

**Parameters**

- **nbunch**: iterable container, optional (default=all nodes)
  A container of nodes. The container will be iterated through once.

- **data**: bool, optional (default=False)
  If True, return edge attribute dict with each edge.

- **keys**: bool, optional (default=False)
  If True, return edge keys with each edge.

**Returns**

- **edge_iter**: iterator
  An iterator of (u,v), (u,v,d) or (u,v,key,d) tuples of edges.

**See Also**
edges return a list of edges

Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored. For directed graphs this returns the out-edges.

Examples

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

in_edges

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

Parameters

nbunch : iterable container, optional (default= all nodes)
A container of nodes. The container will be iterated through once.

data : bool, optional (default=False)
If True, return edge attribute dict with each edge.

keys : bool, optional (default=False)
If True, return edge keys with each edge.

Returns

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

See Also:

out_edges return a list of outgoing edges

in_edges_iter

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

Parameters

nbunch : iterable container, optional (default= all nodes)
A container of nodes. The container will be iterated through once.

data : bool, optional (default=False)
If True, return edge attribute dict with each edge.

keys : bool, optional (default=False)
If True, return edge keys with each edge.

**Returns** in_edge_iter : iterator
   An iterator of (u,v), (u,v,d) or (u,v,key,d) tuples of edges.

**See Also:**

edges_iter return an iterator of edges

g**et_edge_data**

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

**Parameters** u,v : nodes
default: any Python object (default=None)
   Value to return if the edge (u,v) is not found.
key : hashable identifier, optional (default=None)
   Return data only for the edge with specified key.

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

**Notes**

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

```python
>>> 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.

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

**Examples**

```python
>>> 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
```
neighbors

MultiDiGraph.neighbors(n)
  Return a list of successor nodes of n.
  neighbors() and successors() are the same function.

neighbors_iter

MultiDiGraph.neighbors_iter(n)
  Return an iterator over successor nodes of n.
  neighbors_iter() and successors_iter() are the same.

__getitem__

MultiDiGraph.__getitem__(n)
  Return a dict of neighbors of node n. Use the expression ‘G[n]’.
  Parameters n : node
    A node in the graph.
  Returns adj_dict : dictionary
    The adjacency dictionary for nodes connected to n.

Notes

G[n] is similar to G.neighbors(n) but the internal data dictionary is returned instead of a list.
Assigning G[n] will corrupt the internal graph data structure. Use G[n] for reading data only.

Examples

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

successors

MultiDiGraph.successors(n)
  Return a list of successor nodes of n.
  neighbors() and successors() are the same function.

successors_iter

MultiDiGraph.successors_iter(n)
  Return an iterator over successor nodes of n.
  neighbors_iter() and successors_iter() are the same.
**predecessors**

`MultiDiGraph.predecessors(n)`

Return a list of predecessor nodes of \( n \).

**predecessors_iter**

`MultiDiGraph.predecessors_iter(n)`

Return an iterator over predecessor nodes of \( n \).

**adjacency_list**

`MultiDiGraph.adjacency_list()`

Return an adjacency list representation of the graph.

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

**Returns** `adj_list`: lists of lists

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

**See Also:** `adjacency_iter`

**Examples**

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

**adjacency_iter**

`MultiDiGraph.adjacency_iter()`

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

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

**Returns** `adj_iter`: iterator

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

**See Also:** `adjacency_list`

**Examples**

```python
>>> 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: {}})]
```
MultiDiGraph.nbunch_iter(nbunch=None)

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

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

Parameters
nbunch : iterable container, optional (default=all nodes)
A container of nodes. The container will be iterated through once.

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

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

See Also:
Graph.__iter__

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

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

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

Information about graph structure

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MultiDiGraph.has_node(n)</td>
<td>Return True if the graph contains the node n.</td>
</tr>
<tr>
<td>MultiDiGraph.<strong>contains</strong>(n)</td>
<td>Return True if n is a node, False otherwise. Use the expression</td>
</tr>
<tr>
<td>MultiDiGraph.has_edge(u, v[, key])</td>
<td>Return True if the graph has an edge between nodes u and v.</td>
</tr>
<tr>
<td>MultiDiGraph.order()</td>
<td>Return the number of nodes in the graph.</td>
</tr>
<tr>
<td>MultiDiGraph.number_of_nodes()</td>
<td>Return the number of nodes in the graph.</td>
</tr>
<tr>
<td>MultiDiGraph.<strong>len</strong>()</td>
<td>Return the number of nodes.</td>
</tr>
<tr>
<td>MultiDiGraph.degree([nbunch, weight])</td>
<td>Return the degree of a node or nodes.</td>
</tr>
<tr>
<td>MultiDiGraph.degree_iter([nbunch, weight])</td>
<td>Return an iterator for (node, degree).</td>
</tr>
<tr>
<td>MultiDiGraph.in_degree([nbunch, weight])</td>
<td>Return the in-degree of a node or nodes.</td>
</tr>
<tr>
<td>MultiDiGraph.in_degree_iter([nbunch, weight])</td>
<td>Return an iterator for (node, in-degree).</td>
</tr>
<tr>
<td>MultiDiGraph.out_degree([nbunch, weight])</td>
<td>Return the out-degree of a node or nodes.</td>
</tr>
<tr>
<td>MultiDiGraph.out_degree_iter([nbunch, weight])</td>
<td>Return an iterator for (node, out-degree).</td>
</tr>
<tr>
<td>MultiDiGraph.size([weight])</td>
<td>Return the number of edges.</td>
</tr>
<tr>
<td>MultiDiGraph.number_of_edges([u, v])</td>
<td>Return the number of edges between two nodes.</td>
</tr>
<tr>
<td>MultiDiGraph.nodes_with_selfloops()</td>
<td>Return a list of nodes with self loops.</td>
</tr>
<tr>
<td>MultiDiGraph.selfloop_edges([data, keys])</td>
<td>Return a list of selfloop edges.</td>
</tr>
<tr>
<td>MultiDiGraph.number_of_selfloops()</td>
<td>Return the number of selfloop edges.</td>
</tr>
</tbody>
</table>
**has_node**

MultDiGraph.\texttt{has\_node}(n)

Return True if the graph contains the node \(n\).

**Parameters**

\(n\) : node

**Examples**

```python
>>> 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

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

**__contains__**

MultDiGraph.\texttt{__contains__}(n)

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

**Examples**

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

**has_edge**

MultDiGraph.\texttt{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.

\(\text{key}\) : hashable identifier, optional (default=None)

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

**Returns**

\(\text{edge\_ind}\) : 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

order

MultiDiGraph.order()
Return the number of nodes in the graph.

Returns  nnodes : int
The number of nodes in the graph.

See Also:
number_of_nodes, __len__

number_of_nodes

MultiDiGraph.number_of_nodes()
Return the number of nodes in the graph.

Returns  nnodes : int
The number of nodes in the graph.

See Also:
order, __len__

Examples

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

MultiDiGraph.__len__()

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

Returns nnodes : int

The number of nodes in the graph.

Examples

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

degree

MultiDiGraph.degree (nbunch=None, weight=None)

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.

weight : string or None, optional (default=None)

The edge attribute that holds the numerical value used as a weight. If None, then each
edge has weight 1. The degree is the sum of the 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]

degree_iter

MultiDiGraph.degree_iter (nbunch=None, weight=None)

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.

weight : string or None, optional (default=None)

The edge attribute that holds the numerical value used as a weight. If None, then each
edge has weight 1. The degree is the sum of the edge weights.

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)]

in_degree

MultiDiGraph.in_degree(nbunch=None, weight=None)

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.

weight : string or None, optional (default=None)

The edge attribute that holds the numerical value used as a weight. If None, then each
edge has weight 1. The degree is the sum of the 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]
**in_degree_iter**

`MultiDiGraph.in_degree_iter(nbunch=None, weight=None)`  
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.  
`weight` : string or None, optional (default=None)  
The edge attribute that holds the numerical value used as a weight. If None, then each  
edge has weight 1. The degree is the sum of the 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**

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

**out_degree**

`MultiDiGraph.out_degree(nbunch=None, weight=None)`  
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.  
`weight` : string or None, optional (default=None)  
The edge attribute that holds the numerical value used as a weight. If None, then each  
edge has weight 1. The degree is the sum of the 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**

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

out_degree_iter

MultiDiGraph.out_degree_iter (nbunch=None, weight=None)
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.

weight : string or None, optional (default=None)
The edge attribute that holds the numerical value used as a weight. If None, then each
edge has weight 1. The degree is the sum of the edge weights.

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)]

size

MultiDiGraph.size (weight=None)
Return the number of edges.

Parameters

weight : string or None, optional (default=None)
The edge attribute that holds the numerical value used as a weight. If None, then each
edge has weight 1.

Returns

nedges : int
The number of edges or sum of edge weights in the graph.

See Also:
number_of_edges

3.2. Basic graph types 121
Examples

```python
>>> 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(weight='weight')
6.0
```

number_of_edges

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

Return the number of edges between two nodes.

Parameters:
- `u`: 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

```python
>>> 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
```

nodes_with_selfloops

MultiDiGraph.nodes_with_selfloops()

Return a list of nodes with self loops.

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

Returns:
- `nodelist`: list
  A list of nodes with self loops.
See Also:

selfloop_edges, number_of_selfloops

Examples

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

**selfloop_edges**

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

Return a list of selfloop edges.

A selfloop edge has the same node at both ends.

**Parameters**

- `data` : bool, optional (default=False)
  - Return selfloop edges as two tuples (u,v) (data=False) or three-tuples (u,v,data) (data=True)

- `keys` : bool, optional (default=False)
  - If True, return edge keys with each edge.

**Returns**

- `edgelist` : list of edge tuples
  - A list of all selfloop edges.

See Also:

nodes_with_selfloops, number_of_selfloops

Examples

```python
>>> 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, {})]
```

**number_of_selfloops**

`MultiDiGraph.number_of_selfloops()`

Return the number of selfloop edges.

A selfloop edge has the same node at both ends.
Returns nloops : int

The number of selfloops.

See Also:

nodes_with_selfloops, selfloop_edges

Examples

```python
>>> 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

<table>
<thead>
<tr>
<th>MultiDiGraph.copy()</th>
<th>Return a copy of the graph.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MultiDiGraph.to_undirected([reciprocal])</td>
<td>Return an undirected representation of the digraph.</td>
</tr>
<tr>
<td>MultiDiGraph.to_directed()</td>
<td>Return a directed copy of the graph.</td>
</tr>
<tr>
<td>MultiDiGraph.subgraph(nbunch)</td>
<td>Return the subgraph induced on nodes in nbunch.</td>
</tr>
<tr>
<td>MultiDiGraph.reverse([copy])</td>
<td>Return the reverse of the graph.</td>
</tr>
</tbody>
</table>

**copy**

MultiDiGraph.copy()

Return a copy of the graph.

Returns G : Graph

A copy of the graph.

See Also:

to_directed return a directed copy of the graph.

Notes

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

Examples

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

**to_undirected**

MultiDiGraph.to_undirected(reciprocal=False)

Return an undirected representation of the digraph.
**Parameters** reciprocal : bool (optional)

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

**Returns** G : MultiGraph

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

**Notes**

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

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

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

**Examples**

```python
>>> 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.

---

**to_directed**

MultiDiGraph.to_directed()

Return a directed copy of the graph.

**Returns** G : MultiDiGraph

A deepcopy of the graph.

**Notes**

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

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

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

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

---

3.2. Basic graph types
```python
>>> G = nx.MultiDiGraph()
>>> G.add_path([0,1])
>>> H = G.to_directed()
>>> H.edges()
[(0, 1)]
```

**subgraph**

*MultiDiGraph.*

```
subgraph(nbunch)
```

Return the subgraph induced on nodes in `nbunch`.

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

**Parameters**

`nbunch` : list, iterable

A container of nodes which will be iterated through once.

**Returns**

`G` : Graph

A subgraph of the graph with the same edge attributes.

**Notes**

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

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

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

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

**Examples**

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

**reverse**

*MultiDiGraph.*

```
reverse(copy=True)
```

Return the reverse of the graph.

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

**Parameters**

`copy` : bool optional (default=True)

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

4.1.1 Clique

Clique in an undirected graph $G = (V, E)$ is a subset of the vertex set $C \subseteq V$, such that for every two vertices in $C$, there exists an edge connecting the two. This is equivalent to saying that the subgraph induced by $C$ is complete (in some cases, the term clique may also refer to the subgraph).

A maximum clique is a clique of the largest possible size in a given graph. The clique number $\omega(G)$ of a graph $G$ is the number of vertices in a maximum clique in $G$. The intersection number of $G$ is the smallest number of cliques that together cover all edges of $G$.

http://en.wikipedia.org/wiki/Maximum_clique

References

[R139]
clique_removal

**clique_removal** \((G)\)

Repeatedly remove cliques from the graph. Results in a \(O(|V|/\log |V|)^2\) approximation of maximum clique & independent set. Returns the largest independent set found, along with found maximal cliques.

**Parameters**

- \(G\) : NetworkX graph
  - Undirected graph

**Returns**

- **max_ind_cliques** : (set, list) tuple
  - Maximal independent set and list of maximal cliques (sets) in the graph.

**References**

[R138]

### 4.1.2 Clustering

**average_clustering** \((G[, \text{trials}])\) Estimates the average clustering coefficient of \(G\).

**average_clustering** \((G, \text{trials}=1000)\) Estimates the average clustering coefficient of \(G\).

The local clustering of each node in \(G\) is the fraction of triangles that actually exist over all possible triangles in its neighborhood. The average clustering coefficient of a graph \(G\) is the mean of local clusterings.

This function finds an approximate average clustering coefficient for \(G\) by repeating \(n\) times (defined in \(\text{trials}\)) the following experiment: choose a node at random, choose two of its neighbors at random, and check if they are connected. The approximate coefficient is the fraction of triangles found over the number of trials [R140].

**Parameters**

- \(G\) : NetworkX graph
- \(\text{trials}\) : integer
  - Number of trials to perform (default 1000).

**Returns**

- **c** : float
  - Approximated average clustering coefficient.

**References**

[R140]

### 4.1.3 Dominating Set

A dominating set for a graph \(G = (V, E)\) is a subset \(D\) of \(V\) such that every vertex not in \(D\) is joined to at least one member of \(D\) by some edge. The domination number \(\gamma(G)\) is the number of vertices in a smallest dominating set for \(G\). Given a graph \(G = (V, E)\) find a minimum weight dominating set \(V'\).
An edge dominating set for a graph $G = (V, E)$ is a subset $D$ of $E$ such that every edge not in $D$ is adjacent to at least one edge in $D$.

**min_weighted_dominating_set**

```python
min_weighted_dominating_set(G[, weight])  # Return minimum weight vertex dominating set.
```

**min_edge_dominating_set**

```python
min_edge_dominating_set(G)  # Return minimum cardinality edge dominating set.
```

**Notes**

This algorithm computes an approximate minimum weighted dominating set for the graph $G$. The upper-bound on the size of the solution is $\log w(V) \times OPT$. Runtime of the algorithm is $O(|E|)$.

**References**

[R141]

**min_edge_dominating_set**

```python
min_edge_dominating_set(G)  # Return minimum cardinality edge dominating set.
```

**Notes**

The algorithm computes an approximate solution to the edge dominating set problem. The result is no more than $2 \times OPT$ in terms of size of the set. Runtime of the algorithm is $O(|E|)$.
4.1.4 Independent Set

Independent Set

Independent set or stable set is a set of vertices in a graph, no two of which are adjacent. That is, it is a set I of vertices such that for every two vertices in I, there is no edge connecting the two. Equivalently, each edge in the graph has at most one endpoint in I. The size of an independent set is the number of vertices it contains.

A maximum independent set is a largest independent set for a given graph G and its size is denoted \( \alpha(G) \). The problem of finding such a set is called the maximum independent set problem and is an NP-hard optimization problem. As such, it is unlikely that there exists an efficient algorithm for finding a maximum independent set of a graph.


Independent set algorithm is based on the following paper:


\[
\text{maximum Independent set}(G) \quad \text{Return an approximate maximum independent set.}
\]

minimum maximal_matching

minimum maximal_matching(\( G \))

Returns the minimum maximal matching of \( G \). That is, out of all maximal
**min_maximal_matching**

**min_maximal_matching**(G)

Returns the minimum maximal matching of G. That is, out of all maximal matchings of the graph G, the smallest is returned.

**Parameters**

- G : NetworkX graph
  - Undirected graph

**Returns**

- min_maximal_matching : set
  - Returns a set of edges such that no two edges share a common endpoint and every edge not in the set shares some common endpoint in the set. Cardinality will be 2*OPT in the worst case.

**Notes**

The algorithm computes an approximate solution for the minimum maximal cardinality matching problem. The solution is no more than 2 * OPT in size. Runtime is \(O(|E|)\).

**References**

[R143]

**4.1.6 Ramsey**

Ramsey numbers.

- \(\text{ramsey}_R2(G)\) Approximately computes the Ramsey number \(R(2; s, t)\) for graph.

**ramsey_R2**

**ramsey_R2**(G)

Approximately computes the Ramsey number \(R(2; s, t)\) for graph.

**Parameters**

- G : NetworkX graph
  - Undirected graph

**Returns**

- max_pair : (set, set) tuple
  - Maximum clique, Maximum independent set.

**4.1.7 Vertex Cover**

Given an undirected graph \(G = (V, E)\) and a function \(w\) assigning nonnegative weights to its vertices, find a minimum weight subset of \(V\) such that each edge in \(E\) is incident to at least one vertex in the subset.

http://en.wikipedia.org/wiki/Vertex_cover

- \(\text{min_weighted_vertex_cover}(G[, \text{weight}])\) 2-OPT Local Ratio for Minimum Weighted Vertex Cover
min_weighted_vertex_cover

\texttt{min\_weighted\_vertex\_cover} \hspace{1em} (\textit{G}, \textit{weight}=\textit{None})

\begin{itemize}
  \item 2-OPT Local Ratio for Minimum Weighted Vertex Cover
  \item Find an approximate minimum weighted vertex cover of a graph.
\end{itemize}

**Parameters**

- \textit{G} : NetworkX graph
  
  Undirected graph

- \textit{weight} : None or string, optional (default = None)
  
  If None, every edge has weight/distance/cost 1. If a string, use this edge attribute as the
  edge weight. Any edge attribute not present defaults to 1.

**Returns**

- \textit{min\_weighted\_cover} : set
  
  Returns a set of vertices whose weight sum is no more than 2 * OPT.

**Notes**

Local-Ratio algorithm for computing an approximate vertex cover. Algorithm greedily reduces the costs over
edges and iteratively builds a cover. Worst-case runtime is $O(|E|)$.

**References**

[R144]

\section*{4.2 Assortativity}

\subsection*{4.2.1 Assortativity}

\begin{verbatim}
degree_assortativity_coefficient(G[, x, y, ...]) Compute degree assortativity of graph.
attribute_assortativity_coefficient(G, attribute) Compute assortativity for node attributes.
numeric_assortativity_coefficient(G, attribute) Compute assortativity for numerical node attributes.
degree_pearson_correlation_coefficient(G[, ...]) Compute degree assortativity of graph.
\end{verbatim}

degree_assortativity_coefficient

\texttt{degree\_assortativity\_coefficient} \hspace{1em} (\textit{G}, \textit{x}=\textit{‘out’}, \textit{y}=\textit{‘in’}, \textit{weight}=\textit{None}, \textit{nodes}=\textit{None})

\begin{itemize}
  \item Compute degree assortativity of graph.
\end{itemize}

Assortativity measures the similarity of connections in the graph with respect to the node degree.

**Parameters**

- \textit{G} : NetworkX graph
  
  \begin{itemize}
  \item \textit{x} : string (\textit{‘in’}, \textit{‘out’})
    
    The degree type for source node (directed graphs only).
  \item \textit{y} : string (\textit{‘in’}, \textit{‘out’})
  \end{itemize}
The degree type for target node (directed graphs only).

**weight**: string or None, optional (default=None)

The edge attribute that holds the numerical value used as a weight. If None, then each edge has weight 1. The degree is the sum of the edge weights adjacent to the node.

**nodes**: list or iterable (optional)

Compute degree assortativity only for nodes in container. The default is all nodes.

**Returns**  \( r \) : float

Assortativity of graph by degree.

**See Also**

attribute_assortativity_coefficient, numeric_assortativity_coefficient, neighbor_connectivity, degree_mixing_dict, degree_mixing_matrix

**Notes**

This computes Eq. (21) in Ref. [R148], where \( e \) is the joint probability distribution (mixing matrix) of the degrees. If \( G \) is directed then the matrix \( e \) is the joint probability of the user-specified degree type for the source and target.

**References**

[R148], [R149]

**Examples**

```python
>>> G=nx.path_graph(4)
>>> r=nx.degree_assortativity_coefficient(G)
>>> print("%3.1f"%r)
-0.5
```

**attribute_assortativity_coefficient**

**attribute_assortativity_coefficient** \( (G, attribute, nodes=None) \)

Compute assortativity for node attributes.

Assortativity measures the similarity of connections in the graph with respect to the given attribute.

**Parameters**

\( G \) : NetworkX graph

attribute : string

Node attribute key

nodes: list or iterable (optional)

Compute attribute assortativity for nodes in container. The default is all nodes.

**Returns**

\( r \) : float

Assortativity of graph for given attribute
Notes

This computes Eq. (2) in Ref. [R145], \( \frac{\text{trace}(M)-\sum(M)}{1-\sum(M)} \), where \( M \) is the joint probability distribution (mixing matrix) of the specified attribute.

References

[R145]

Examples

```python
>>> 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_coefficient(G, 'color'))
1.0
```

numeric_assortativity_coefficient

**numeric_assortativity_coefficient** (*G*, *attribute*, *nodes=None*)

Compute assortativity for numerical node attributes.

Assortativity measures the similarity of connections in the graph with respect to the given numeric attribute.

**Parameters**

- **G**: NetworkX graph
- **attribute**: string
  - Node attribute key
- **nodes**: list or iterable (optional)
  - Compute numeric assortativity only for attributes of nodes in container. The default is all nodes.

**Returns**

- **r**: float
  - Assortativity of graph for given attribute

Notes

This computes Eq. (21) in Ref. [R153], for the mixing matrix of the specified attribute.

References

[R153]

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_coefficient(G,'size'))
1.0

degree_pearson_correlation_coefficient

degree_pearson_correlation_coefficient (G, x='out', y='in', weight=None, nodes=None)
Compute degree assortativity of graph.
Assortativity measures the similarity of connections in the graph with respect to the node degree.
This is the same as degree_assortativity_coefficient but uses the potentially faster scipy.stats.pearsonr function.

Parameters  

G : NetworkX graph
  The graph whose degree Pearson correlation coefficient is calculated
x : string ('in','out')
  The degree type for source node (directed graphs only).
y : string ('in','out')
  The degree type for target node (directed graphs only).
weight : string or None, optional (default=None)
  The edge attribute that holds the numerical value used as a weight. If None, then each edge has weight 1. The degree is the sum of the edge weights adjacent to the node.
nodes : list or iterable (optional)
  Compute pearson correlation of degrees only for specified nodes. The default is all nodes.

Returns  
r : float
  Assortativity of graph by degree.

Notes

This calls scipy.stats.pearsonr.

References

[R150], [R151]

Examples

>>> G=nx.path_graph(4)
>>> r=nx.degree_pearson_correlation_coefficient(G)
>>> print("%.3f"%r)
-0.5

4.2. Assortativity

135
4.2.2 Average neighbor degree

\texttt{average_neighbor_degree}(G[, source, target, ...])  Returns the average degree of the neighborhood of each node.

\texttt{average_neighbor_degree}

\texttt{average_neighbor_degree} \hspace{1em} (G, source='out', target='out', nodes=None, weight=None)

Returns the average degree of the neighborhood of each node.

The average degree of a node \( i \) is

\[
\bar{k}_{nn,i} = \frac{1}{|N(i)|} \sum_{j \in N(i)} k_j
\]

where \( N(i) \) are the neighbors of node \( i \) and \( k_j \) is the degree of node \( j \) which belongs to \( N(i) \). For weighted graphs, an analogous measure can be defined [R147],

\[
\bar{k}_{nn,i}^w = \frac{1}{s_i} \sum_{j \in N(i)} w_{ij}k_j
\]

where \( s_i \) is the weighted degree of node \( i \), \( w_{ij} \) is the weight of the edge that links \( i \) and \( j \) and \( N(i) \) are the neighbors of node \( i \).

Parameters  
\begin{itemize}
  \item \texttt{G} : NetworkX graph
  \item \texttt{source} : string ("in"|"out")  
    Directed graphs only. Use "in"- or "out"-degree for source node.
  \item \texttt{target} : string ("in"|"out")
    Directed graphs only. Use "in"- or "out"-degree for target node.
  \item \texttt{nodes} : list or iterable, optional
    Compute neighbor degree for specified nodes. The default is all nodes in the graph.
  \item \texttt{weight} : string or None, optional (default=None)
    The edge attribute that holds the numerical value used as a weight. If None, then each edge has weight 1.
\end{itemize}

Returns  
\texttt{d} : dict

A dictionary keyed by node with average neighbors degree value.

See Also:
\texttt{average_degree_connectivity}

Notes

For directed graphs you can also specify in-degree or out-degree by passing keyword arguments.

References

[R147]
Examples

```python
>>> G=nx.path_graph(4)
>>> G.edge[0][1]['weight'] = 5
>>> G.edge[2][3]['weight'] = 3

>>> nx.average_neighbor_degree(G)
{0: 2.0, 1: 1.5, 2: 1.5, 3: 2.0}

>>> nx.average_neighbor_degree(G, weight='weight')
{0: 2.0, 1: 1.1666666666666667, 2: 1.25, 3: 2.0}

>>> G=nx.DiGraph()
>>> G.add_path([0,1,2,3])

>>> nx.average_neighbor_degree(G, source='in', target='in')
{0: 1.0, 1: 1.0, 2: 1.0, 3: 0.0}

>>> nx.average_neighbor_degree(G, source='out', target='out')
{0: 1.0, 1: 1.0, 2: 0.0, 3: 0.0}
```

4.2.3 Average degree connectivity

```python
average_degree_connectivity(G[, source, ...])  Compute the average degree connectivity of graph.

k_nearest_neighbors(G[, source, target, ...])  Compute the average degree connectivity of graph.
```

average_degree_connectivity

The average degree connectivity is the average nearest neighbor degree of nodes with degree k. For weighted
graphs, an analogous measure can be computed using the weighted average neighbors degree defined in [R146],
for a node i, as:

\[ k_{nn,i}^w = \frac{1}{s_i} \sum_{j \in N(i)} w_{ij}k_j \]

where \( s_i \) is the weighted degree of node \( i \), \( w_{ij} \) is the weight of the edge that links \( i \) and \( j \), and \( N(i) \) are the
neighbors of node \( i \).

Parameters  G : NetworkX graph

source : “in”|”out”|”in+out” (default:”in+out”)  Directed graphs only. Use “in”- or “out”-degree for source node.

target : “in”|”out”|”in+out” (default:”in+out”)  Directed graphs only. Use “in”- or “out”-degree for target node.

nodes: list or iterable (optional)  Compute neighbor connectivity for these nodes. The default is all nodes.

weight : string or None, optional (default=None)  The edge attribute that holds the numerical value used as a weight. If None, then each
edge has weight 1.
Returns  

d: dict

A dictionary keyed by degree k with the value of average connectivity.

See Also:

neighbors_average_degree

Notes

This algorithm is sometimes called “k nearest neighbors” and is also available as k_{n earestneighbors}.

References

[R146]

Examples

```python
>>> G=nx.path_graph(4)
>>> G.edge[1][2][’weight’] = 3
>>> nx.k_nearest_neighbors(G)
(1: 2.0, 2: 1.5)
>>> nx.k_nearest_neighbors(G, weight=’weight’)
(1: 2.0, 2: 1.75)
```

k_nearest_neighbors

k_nearest_neighbors(G, source=’in+out’, target=’in+out’, nodes=None, weight=None)

Compute the average degree connectivity of graph.

The average degree connectivity is the average nearest neighbor degree of nodes with degree k. For weighted graphs, an analogous measure can be computed using the weighted average neighbors degree defined in [R152], for a node i, as:

\[ k_{nn, i}^w = \frac{1}{s_i} \sum_{j \in N(i)} w_{ij} k_j \]

where \( s_i \) is the weighted degree of node \( i \), \( w_{ij} \) is the weight of the edge that links \( i \) and \( j \), and \( N(i) \) are the neighbors of node \( i \).

Parameters  

G : NetworkX graph

source : “in”|”out”|”in+out” (default:”in+out”)  

Directed graphs only. Use “in”- or “out”-degree for source node.

target : “in”|”out”|”in+out” (default:”in+out”)  

Directed graphs only. Use “in”- or “out”-degree for target node.

nodes: list or iterable (optional)

Compute neighbor connectivity for these nodes. The default is all nodes.

weight : string or None, optional (default=None)
The edge attribute that holds the numerical value used as a weight. If None, then each edge has weight 1.

**Returns**  
d: dict  
A dictionary keyed by degree k with the value of average connectivity.

**See Also:**  
neighbors_average_degree

**Notes**  
This algorithm is sometimes called ‘k nearest neighbors’ and is also available as \( k_{\text{nearest}} \text{neighbors} \).

**References**  
[R152]

**Examples**

```python
>>> G=nx.path_graph(4)
>>> G.edge[1][2]["weight"] = 3
>>> nx.k_nearest_neighbors(G)
{1: 2.0, 2: 1.5}
>>> nx.k_nearest_neighbors(G, weight='weight')
{1: 2.0, 2: 1.75}
```

### 4.2.4 Mixing

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>attribute_mixing_matrix(G, attribute[, ...])</code></td>
<td>Return mixing matrix for attribute.</td>
</tr>
<tr>
<td><code>degree_mixing_matrix(G[, x, y, weight, ...])</code></td>
<td>Return mixing matrix for attribute.</td>
</tr>
<tr>
<td><code>degree_mixing_dict(G[, x, y, weight, nodes, ...])</code></td>
<td>Return dictionary representation of mixing matrix for degree.</td>
</tr>
<tr>
<td><code>attribute_mixing_dict(G, attribute[, nodes, ...])</code></td>
<td>Return dictionary representation of mixing matrix for attribute.</td>
</tr>
</tbody>
</table>

**attribute_mixing_matrix**

```python
attribute_mixing_matrix(G, attribute[, nodes=None, mapping=None, normalized=True])
```

Return mixing matrix for attribute.

**Parameters**  

- **G**: graph  
  NetworkX graph object.
- **attribute**: string  
  Node attribute key.
- **nodes**: list or iterable (optional)  
  Use only nodes in container to build the matrix. The default is all nodes.
- **mapping**: dictionary, optional

### 4.2. Assortativity
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.

degree_mixing_matrix

degree_mixing_matrix(G, x='out', y='in', weight=None, nodes=None, normalized=True)

Return mixing matrix for attribute.

Parameters G : graph

NetworkX graph object.

x: string ('in','out')

The degree type for source node (directed graphs only).

y: string ('in','out')

The degree type for target node (directed graphs only).

nodes: list or iterable (optional)

Build the matrix using only nodes in container. The default is all nodes.

weight: string or None, optional (default=None)

The edge attribute that holds the numerical value used as a weight. If None, then each edge has weight 1. The degree is the sum of the edge weights adjacent to the node.

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.

degree_mixing_dict

degree_mixing_dict(G, x='out', y='in', weight=None, nodes=None, normalized=False)

Return dictionary representation of mixing matrix for degree.

Parameters G : graph

NetworkX graph object.

x: string ('in','out')

The degree type for source node (directed graphs only).

y: string ('in','out')

The degree type for target node (directed graphs only).

weight: string or None, optional (default=None)

The edge attribute that holds the numerical value used as a weight. If None, then each edge has weight 1. The degree is the sum of the edge weights adjacent to the node.

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.
**attribute_mixing_dict**

*attribute_mixing_dict* *(G, attribute, nodes=None, normalized=False)*

Return dictionary representation of mixing matrix for attribute.

**Parameters**
- **G**: graph
  - NetworkX graph object.
- **attribute**: string
  - Node attribute key.
- **nodes**: list or iterable (optional)
  - Use nodes in container to build the dict. The default is all nodes.
- **normalized**: bool (default=False)
  - Return counts if False or probabilities if True.

**Returns**
- **d**: dictionary
  - Counts or joint probability of occurrence of attribute pairs.

**Examples**

```python
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
```

### 4.3 Bipartite

This module provides functions and operations for bipartite graphs. Bipartite graphs \( B = (U, V, E) \) have two node sets \( U, V \) and edges in \( E \) that only connect nodes from opposite sets. It is common in the literature to use a spatial analogy referring to the two node sets as top and bottom nodes.

The bipartite algorithms are not imported into the networkx namespace at the top level so the easiest way to use them is with:

```python
>>> import networkx as nx
>>> from networkx.algorithms import bipartite
```
NetworkX does not have a custom bipartite graph class but the `Graph()` or `DiGraph()` classes can be used to represent bipartite graphs. However, you have to keep track of which set each node belongs to, and make sure that there is no edge between nodes of the same set. The convention used in NetworkX is to use a node attribute named “bipartite” with values 0 or 1 to identify the sets each node belongs to.

For example:

```python
>>> B = nx.Graph()
>>> B.add_nodes_from([1,2,3,4], bipartite=0)  # Add the node attribute "bipartite"
>>> B.add_nodes_from(['a','b','c'], bipartite=1)
>>> B.add_edges_from(((1,'a'), (1,'b'), (2,'b'), (2,'c'), (3,'c'), (4,'a')))
```

Many algorithms of the bipartite module of NetworkX require, as an argument, a container with all the nodes that belong to one set, in addition to the bipartite graph $B$. If $B$ is connected, you can find the node sets using a two-coloring algorithm:

```python
>>> nx.is_connected(B)
True
>>> bottom_nodes, top_nodes = bipartite.sets(B)
```

`list(top_nodes)` [1, 2, 3, 4] `list(bottom_nodes)` ['a', 'c', 'b']

However, if the input graph is not connected, there are more than one possible colorations. Thus, the following result is correct:

```python
>>> B.remove_edge(2,'c')
>>> nx.is_connected(B)
False
>>> bottom_nodes, top_nodes = bipartite.sets(B)
```

`list(top_nodes)` [1, 2, 4, 'c'] `list(bottom_nodes)` ['a', 3, 'b']

Using the “bipartite” node attribute, you can easily get the two node sets:

```python
>>> top_nodes = set(n for n,d in B.nodes(data=True) if d['bipartite']==0)
>>> bottom_nodes = set(B) - top_nodes
```

`list(top_nodes)` [1, 2, 3, 4] `list(bottom_nodes)` ['a', 'c', 'b']

So you can easily use the bipartite algorithms that require, as an argument, a container with all nodes that belong to one node set:

```python
>>> print(round(bipartite.density(B, bottom_nodes),2))
0.42
>>> G = bipartite.projected_graph(B, top_nodes)
>>> G.edges()
[(1, 2), (1, 4)]
```

All bipartite graph generators in NetworkX build bipartite graphs with the “bipartite” node attribute. Thus, you can use the same approach:

```python
>>> RB = nx.bipartite_random_graph(5, 7, 0.2)
>>> RB_top = set(n for n,d in RB.nodes(data=True) if d['bipartite']==0)
>>> RB_bottom = set(RB) - RB_top
>>> list(RB_top)
[0, 1, 2, 3, 4]
>>> list(RB_bottom)
[5, 6, 7, 8, 9, 10, 11]
```

For other bipartite graph generators see the bipartite section of *Graph generators*. 
4.3.1 Basic functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>is_bipartite(G)</code></td>
<td>Returns True if graph G is bipartite, False if not.</td>
</tr>
<tr>
<td><code>is_bipartite_node_set(G, nodes)</code></td>
<td>Returns True if nodes and G/nodes are a bipartition of G.</td>
</tr>
<tr>
<td><code>sets(G)</code></td>
<td>Returns bipartite node sets of graph G.</td>
</tr>
<tr>
<td><code>color(G)</code></td>
<td>Returns a two-coloring of the graph.</td>
</tr>
<tr>
<td><code>density(B, nodes)</code></td>
<td>Return density of bipartite graph B.</td>
</tr>
<tr>
<td><code>degrees(B, nodes[, weight])</code></td>
<td>Return the degrees of the two node sets in the bipartite graph B.</td>
</tr>
<tr>
<td><code>biadjacency_matrix(G, row_order[, ...])</code></td>
<td>Return the biadjacency matrix of the bipartite graph G.</td>
</tr>
</tbody>
</table>

**is_bipartite**

`is_bipartite(G)`

Returns True if graph G is bipartite, False if not.

**Parameters**

- `G` : NetworkX graph

**See Also**

- `color`, `is_bipartite_node_set`

**Examples**

```python
>>> from networkx.algorithms import bipartite
>>> G = nx.path_graph(4)
>>> print(bipartite.is_bipartite(G))
True
```

**is_bipartite_node_set**

`is_bipartite_node_set(G, nodes)`

Returns True if nodes and G/nodes are a bipartition of G.

**Parameters**

- `G` : NetworkX graph
- `nodes` : list or container

**Notes**

Check if nodes are a one of a bipartite set.

For connected graphs the bipartite sets are unique. This function handles disconnected graphs.

**Examples**

```python
>>> from networkx.algorithms import bipartite
>>> G = nx.path_graph(4)
>>> X = set([1,3])
>>> print(bipartite.is_bipartite_node_set(G,X))
True
```
sets

```
sets (G)
Returns bipartite node sets of graph G.

Raises an exception if the graph is not bipartite.

Parameters  G : NetworkX graph

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

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

See Also:

color

Examples

```
>>> from networkx.algorithms import bipartite
>>> G = nx.path_graph(4)
>>> X, Y = bipartite.sets(G)
>>> list(X)
[0, 2]
>>> list(Y)
[1, 3]
```

color

color (G)
Returns a two-coloring of the graph.

Raises an exception if the graph is not bipartite.

Parameters  G : NetworkX graph

Returns  color : dictionary

A dictionary keyed by node with a 1 or 0 as data for each node color.

Raises  NetworkXError if the graph is not two-colorable.

Examples

```
>>> from networkx.algorithms import bipartite
>>> G = nx.path_graph(4)
>>> c = bipartite.color(G)
>>> print(c)
{0: 1, 1: 0, 2: 1, 3: 0}
```

You can use this to set a node attribute indicating the bipartite set:

```
>>> nx.set_node_attributes(G, 'bipartite', c)
>>> print(G.node[0]['bipartite'])
1
>>> print(G.node[1]['bipartite'])
0
```
density

density\(B, nodes\)
Return density of bipartite graph B.

Parameters
\[G\] : NetworkX graph

\[nodes\] : list or container
Nodes in one set of the bipartite graph.

Returns \[d\] : float
The bipartite density

See Also:
color

Examples

```python
>>> from networkx.algorithms import bipartite
>>> G = nx.complete_bipartite_graph(3,2)
>>> X=set([0,1,2])
>>> bipartite.density(G,X)
1.0
>>> Y=set([3,4])
>>> bipartite.density(G,Y)
1.0
```

degrees

degrees\(B, nodes, weight=None\)
Return the degrees of the two node sets in the bipartite graph B.

Parameters
\[G\] : NetworkX graph

\[nodes\] : list or container
Nodes in one set of the bipartite graph.

weight : string or None, optional (default=None)
The edge attribute that holds the numerical value used as a weight. If None, then each edge has weight 1. The degree is the sum of the edge weights adjacent to the node.

Returns \[(degX,degY)\] : tuple of dictionaries
The degrees of the two bipartite sets as dictionaries keyed by node.

See Also:
color, density

Examples
>>> from networkx.algorithms import bipartite
>>> G = nx.complete_bipartite_graph(3,2)
>>> Y=set([3,4])
>>> degX,degY=bipartite.degrees(G,Y)
>>> degX
{0: 2, 1: 2, 2: 2}

biadjacency_matrix

**biadjacency_matrix** *(G, row_order, column_order=None, weight='weight', dtype=None)*

Return the biadjacency matrix of the bipartite graph G.

Let \(G = (U, V, E)\) be a bipartite graph with node sets \(U = \{u_1, \ldots, u_r\}\) and \(V = \{v_1, \ldots, v_s\}\). The biadjacency matrix [1] is the \(r \times s\) matrix \(B\) in which \(b_{i,j} = 1\) if, and only if, \((u_i, v_j) \in E\). If the parameter *weight* is not *None* and matches the name of an edge attribute, its value is used instead of 1.

**Parameters**

- **G** : graph
  
  A NetworkX graph

- **row_order** : list of nodes
  
  The rows of the matrix are ordered according to the list of nodes.

- **column_order** : list, optional
  
  The columns of the matrix are ordered according to the list of nodes. If *column_order* is None, then the ordering of columns is arbitrary.

- **weight** : string or None, optional (default='weight')
  
  The edge data key used to provide each value in the matrix. If None, then each edge has weight 1.

- **dtype** : NumPy data type, optional
  
  A valid single NumPy data type used to initialize the array. This must be a simple type such as int or numpy.float64 and not a compound data type (see to_numpy_recarray) If None, then the NumPy default is used.

**Returns**

**B** : numpy matrix

Biadjacency matrix representation of the bipartite graph G.

**See Also:**

to_numpy_matrix, adjacency_matrix

**Notes**

No attempt is made to check that the input graph is bipartite.

For directed bipartite graphs only successors are considered as neighbors. To obtain an adjacency matrix with ones (or weight values) for both predecessors and successors you have to generate two biadjacency matrices where the rows of one of them are the columns of the other, and then add one to the transpose of the other.

**References**

### 4.3.2 Projections

One-mode (unipartite) projections of bipartite graphs.

- **projected_graph**
  
  ```python
  projected_graph(B, nodes[, multigraph])
  ```
  
  Returns the projection of B onto one of its node sets.

  - **B**: NetworkX graph
    
    The input graph should be bipartite.

  - **nodes**: list or iterable
    
    Nodes to project onto (the “bottom” nodes).

  - **multigraph**: bool (default=False)
    
    If True return a multigraph where the multiple edges represent multiple shared neighbors. They edge key in the multigraph is assigned to the label of the neighbor.

  Returns **Graph**: NetworkX graph or multigraph

  A graph that is the projection onto the given nodes.

  See Also:

  - is_bipartite, is_bipartite_node_set, sets, weighted_projected_graph, 
  - collaboration_weighted_projected_graph, overlap_weighted_projected_graph, 
  - generic_weighted_projected_graph

  Notes

  No attempt is made to verify that the input graph B is bipartite. Returns a simple graph that is the projection of the bipartite graph B onto the set of nodes given in list nodes. If multigraph=True then a multigraph is returned with an edge for every shared neighbor.

  Directed graphs are allowed as input. The output will also then be a directed graph with edges if there is a directed path between the nodes.

  The graph and node properties are (shallow) copied to the projected graph.

  Examples

  ```python
  >>> from networkx.algorithms import bipartite
  >>> B = nx.path_graph(4)
  >>> G = bipartite.projected_graph(B, [1,3])
  ```
>>> print(G.nodes())
[1, 3]
>>> print(G.edges())
[(1, 3)]

If nodes \(a\) and \(b\) are connected through both nodes 1 and 2 then building a multigraph results in two edges in the projection onto \([a, b]\):

```python
>>> B = nx.Graph()
>>> B.add_edges_from([('a', 1), ('b', 1), ('a', 2), ('b', 2)])
>>> G = bipartite.projected_graph(B, ['a', 'b'], multigraph=True)
>>> print([{sorted((u,v)) for u,v in G.edges()})
[['a', 'b'], ['a', 'b']]
```

### weighted_projected_graph

**weighted_projected_graph** \((B, nodes, ratio=False)\)

Returns a weighted projection of \(B\) onto one of its node sets.

The weighted projected graph is the projection of the bipartite network \(B\) onto the specified nodes with weights representing the number of shared neighbors or the ratio between actual shared neighbors and possible shared neighbors if \(ratio=True\) \([R163]\). The nodes retain their attributes and are connected in the resulting graph if they have an edge to a common node in the original graph.

**Parameters**

- **B**: NetworkX graph
  - The input graph should be bipartite.

- **nodes**: list or iterable
  - Nodes to project onto (the “bottom” nodes).

- **ratio**: bool (default=False)
  - If True, edge weight is the ratio between actual shared neighbors and possible shared neighbors. If False, edges weight is the number of shared neighbors.

**Returns**

- **Graph**: NetworkX graph
  - A graph that is the projection onto the given nodes.

**See Also:**

- `is_bipartite`, `is_bipartite_node_set`, `sets`, `collaboration_weighted_projected_graph`, `overlap_weighted_projected_graph`, `generic_weighted_projected_graph`, `projected_graph`

**Notes**

No attempt is made to verify that the input graph \(B\) is bipartite. The graph and node properties are (shallow) copied to the projected graph.

**References**

- [R163]
Examples

```python
>>> from networkx.algorithms import bipartite
>>> B = nx.path_graph(4)
>>> G = bipartite.weighted_projected_graph(B, [1,3])
>>> print(G.nodes())
[1, 3]
>>> print(G.edges(data=True))
[(1, 3, {'weight': 1})]
>>> G = bipartite.weighted_projected_graph(B, [1,3], ratio=True)
>>> print(G.edges(data=True))
[(1, 3, {'weight': 0.5})]
```

collaboration_weighted_projected_graph

collaboration_weighted_projected_graph(B, nodes)

Newman’s weighted projection of B onto one of its node sets.

The collaboration weighted projection is the projection of the bipartite network B onto the specified nodes with weights assigned using Newman’s collaboration model [R161]:

\[ w_{v,u} = \sum_{k} \frac{\delta^w_k \delta^v_k}{k_w - 1} \]

where \( v \) and \( u \) are nodes from the same bipartite node set, and \( w \) is a node of the opposite node set. The value \( k_w \) is the degree of node \( w \) in the bipartite network and \( \delta^v_k \) is 1 if node \( v \) is linked to node \( w \) in the original bipartite graph or 0 otherwise.

The nodes retain their attributes and are connected in the resulting graph if have an edge to a common node in the original bipartite graph.

**Parameters**
- **B** : NetworkX graph
  - The input graph should be bipartite.
- **nodes** : list or iterable
  - Nodes to project onto (the “bottom” nodes).

**Returns**
- **Graph** : NetworkX graph
  - A graph that is the projection onto the given nodes.

**See Also:**
- `is_bipartite`, `is_bipartite_node_set`, `sets`, `weighted_projected_graph`, `overlap_weighted_projected_graph`, `generic_weighted_projected_graph`, `projected_graph`

**Notes**

No attempt is made to verify that the input graph B is bipartite. The graph and node properties are (shallow) copied to the projected graph.

**References**

[R161]
Examples

```python
>>> from networkx.algorithms import bipartite
>>> B = nx.path_graph(5)
>>> B.add_edge(1,5)
>>> G = bipartite.collaboration_weighted_projected_graph(B, [0, 2, 4, 5])
>>> print(G.nodes())
[0, 2, 4, 5]
>>> for edge in G.edges(data=True):
... print(edge)
...
(0, 2, {'weight': 0.5})
(0, 5, {'weight': 0.5})
(2, 4, {'weight': 1.0})
(2, 5, {'weight': 0.5})
```

**overlap_weighted_projected_graph**

`overlap_weighted_projected_graph(B, nodes, jaccard=True)`  
Overlap weighted projection of B onto one of its node sets.

The overlap weighted projection is the projection of the bipartite network B onto the specified nodes with weights representing the Jaccard index between the neighborhoods of the two nodes in the original bipartite network [R162]:

\[ w_{v,u} = \frac{|N(u) \cap N(v)|}{|N(u) \cup N(v)|} \]

or if the parameter ‘jaccard’ is False, the fraction of common neighbors by minimum of both nodes degree in the original bipartite graph [R162]:

\[ w_{v,u} = \frac{|N(u) \cap N(v)|}{\min(|N(u)|, |N(v)|)} \]

The nodes retain their attributes and are connected in the resulting graph if have an edge to a common node in the original bipartite graph.

- **Parameters**  
  - `B` : NetworkX graph  
    The input graph should be bipartite.
  - `nodes` : list or iterable  
    Nodes to project onto (the “bottom” nodes).
  - `jaccard` : Bool (default=True)

- **Returns**  
  - `Graph` : NetworkX graph  
    A graph that is the projection onto the given nodes.

- **See Also:**
  - `is_bipartite`, `is_bipartite_node_set`, `sets`, `weighted_projected_graph`, `collaboration_weighted_projected_graph`, `generic_weighted_projected_graph`, `projected_graph`

- **Notes**
  - No attempt is made to verify that the input graph B is bipartite. The graph and node properties are (shallow) copied to the projected graph.
References

[R162]

Examples

```python
>>> from networkx.algorithms import bipartite
>>> B = nx.path_graph(5)
>>> G = bipartite.overlap_weighted_projected_graph(B, [0, 2, 4])
>>> print(G.nodes())
[0, 2, 4]
>>> print(G.edges(data=True))
[(0, 2, {'weight': 0.5}), (2, 4, {'weight': 0.5})]
>>> G = bipartite.overlap_weighted_projected_graph(B, [0, 2, 4], jaccard=False)
>>> print(G.edges(data=True))
[(0, 2, {'weight': 1.0}), (2, 4, {'weight': 1.0})]
```

generic_weighted_projected_graph

```
generic_weighted_projected_graph (B, nodes, weight_function=None)  
Weighted projection of B with a user-specified weight function.

The bipartite network B is projected onto the specified nodes with weights computed by a user-specified function. This function must accept as a parameter the neighborhood sets of two nodes and return an integer or a float.

The nodes retain their attributes and are connected in the resulting graph if they have an edge to a common node in the original graph.

Parameters

- **B**: NetworkX graph
  - The input graph should be bipartite.

- **nodes**: list or iterable
  - Nodes to project onto (the “bottom” nodes).

- **weight_function**: function
  - This function must accept as parameters the same input graph that this function, and two nodes; and return an integer or a float. The default function computes the number of shared neighbors.

Returns **Graph**: NetworkX graph

A graph that is the projection onto the given nodes.

See Also:

- is_bipartite, is_bipartite_node_set, sets, weighted_projected_graph, collaboration_weighted_projected_graph, overlap_weighted_projected_graph, projected_graph

Notes

No attempt is made to verify that the input graph B is bipartite. The graph and node properties are (shallow) copied to the projected graph.

4.3. Bipartite
Examples

```python
>>> from networkx.algorithms import bipartite
>>> # Define some custom weight functions
>>> def jaccard(G, u, v):
...     unbrs = set(G[u])
...     vnbrs = set(G[v])
...     return float(len(unbrs & vnbrs)) / len(unbrs | vnbrs)
... >>> def my_weight(G, u, v, weight='weight'):
...     w = 0
...     for nbr in set(G[u]) & set(G[v]):
...         w += G.edge[u][nbr].get(weight, 1) + G.edge[v][nbr].get(weight, 1)
...     return w
...
>>> # A complete bipartite graph with 4 nodes and 4 edges
>>> B = nx.complete_bipartite_graph(2,2)
>>> # Add some arbitrary weight to the edges
>>> for i,(u,v) in enumerate(B.edges()):
...     B.edge[u][v]['weight'] = i + 1
... for edge in B.edges(data=True):
...     print(edge)
... (0, 2, {'weight': 1})
(0, 3, {'weight': 2})
(1, 2, {'weight': 3})
(1, 3, {'weight': 4})
>>> # Without specifying a function, the weight is equal to # shared partners
>>> G = bipartite.generic_weighted_projected_graph(B, [0, 1])
>>> print(G.edges(data=True))
[(0, 1, {'weight': 2})]
>>> # To specify a custom weight function use the weight_function parameter
>>> G = bipartite.generic_weighted_projected_graph(B, [0, 1], weight_function=jaccard)
>>> print(G.edges(data=True))
[(0, 1, {'weight': 1.0})]
>>> G = bipartite.generic_weighted_projected_graph(B, [0, 1], weight_function=my_weight)
>>> print(G.edges(data=True))
[(0, 1, {'weight': 10})]
```

4.3.3 Spectral

Spectral bipartivity measure.

\[ \text{spectral_bipartivity}(G[, \text{nodes}, \text{weight}]) \quad \text{Returns the spectral bipartivity.} \]

`spectral_bipartivity`

\[ \text{spectral_bipartivity}(G, \text{nodes}=None, \text{weight}='weight') \]

Returns the spectral bipartivity.

- `G`: NetworkX graph
- `nodes`: list or container optional (default is all nodes)

Nodes to return value of spectral bipartivity contribution.
weight : string or None optional (default = ‘weight’)
   Edge data key to use for edge weights. If None, weights set to 1.

Returns  sb : float or dict
   A single number if the keyword nodes is not specified, or a dictionary keyed by node
   with the spectral bipartivity contribution of that node as the value.

See Also:
   color

Notes

This implementation uses Numpy (dense) matrices which are not efficient for storing large sparse graphs.

References

[R165]

Examples

    >>> from networkx.algorithms import bipartite
    >>> G = nx.path_graph(4)
    >>> bipartite.spectral_bipartivity(G)
    1.0

4.3.4 Clustering

    clustering(G[, nodes, mode])  Compute a bipartite clustering coefficient for nodes.
    average_clustering(G[, nodes, mode])  Compute the average bipartite clustering coefficient.
    latapy_clustering(G[, nodes, mode])  Compute a bipartite clustering coefficient for nodes.
    robins_alexander_clustering(G)  Compute the bipartite clustering of G.

clustering

    clustering(G, nodes=None, mode='dot')
    Compute a bipartite clustering coefficient for nodes.

    The bipartite clustering coefficient is a measure of local density of connections defined as [R158]:

        \[ c_u = \frac{\sum_{v \in N(N(u))} c_{uv}}{|N(N(u))|} \]

    where \( N(N(u)) \) are the second order neighbors of \( u \) in \( G \) excluding \( u \), and \( c_{uv} \) is the pairwise clustering coefficient between nodes \( u \) and \( v \).

    The mode selects the function for \( c_{uv} \) which can be:

    dot:

        \[ c_{uv} = \frac{|N(u) \cap N(v)|}{|N(u) \cup N(v)|} \]
\[ c_{uv} = \frac{|N(u) \cap N(v)|}{\min(|N(u)|, |N(v)|)} \]

**Parameters**

- **G**: graph
  - A bipartite graph
- **nodes**: list or iterable (optional)
  - Compute bipartite clustering for these nodes. The default is all nodes in G.
- **mode**: string
  - The pairwise bipartite clustering method to be used in the computation. It must be “dot”, “max”, or “min”.

**Returns**

- **clustering**: dictionary
  - A dictionary keyed by node with the clustering coefficient value.

**See Also**

- `robins_alexander_clustering`, `square_clustering`, `average_clustering`

**References**

[R158]

**Examples**

```python
g = nx.path_graph(4)  # path graphs are bipartite
c = bipartite.clustering(g)
c[0]  # 0.5

# mode='min'
c = bipartite.clustering(g, mode='min')
c[0]  # 1.0
```

**average_clustering**

- **average_clustering** *(G, nodes=None, mode='dot')*
  - Compute the average bipartite clustering coefficient.
  - A clustering coefficient for the whole graph is the average,
    \[
    C = \frac{1}{n} \sum_{v \in G} c_v,
    \]
  - where \(n\) is the number of nodes in \(G\).
Similar measures for the two bipartite sets can be defined [R157]

\[ C_X = \frac{1}{|X|} \sum_{v \in X} c_v, \]

where \( X \) is a bipartite set of \( G \).

**Parameters**

- **G**: graph
  - a bipartite graph
- **nodes**: list or iterable, optional
  - A container of nodes to use in computing the average. The nodes should be either the entire graph (the default) or one of the bipartite sets.
- **mode**: string
  - The pairwise bipartite clustering method. It must be “dot”, “max”, or “min”

**Returns**

- **clustering**: float
  - The average bipartite clustering for the given set of nodes or the entire graph if no nodes are specified.

**See Also:**

- **clustering**

**Notes**

The container of nodes passed to this function must contain all of the nodes in one of the bipartite sets ("top" or "bottom") in order to compute the correct average bipartite clustering coefficients.

**References**

[R157]

**Examples**

```python
>>> from networkx.algorithms import bipartite
>>> G=nx.star_graph(3)  # star graphs are bipartite
>>> bipartite.average_clustering(G)
0.75
>>> X,Y=bipartite.sets(G)
>>> bipartite.average_clustering(G,X)
0.0
>>> bipartite.average_clustering(G,Y)
1.0
```

**latapy_clustering**

**latapy_clustering** \((G, nodes=None, mode='dot')\)

Compute a bipartite clustering coefficient for nodes.
The bipartite clustering coefficient is a measure of local density of connections defined as [R159]:

\[ c_u = \frac{\sum_{v \in N(N(u))} c_{uv}}{|N(N(u))|} \]

where \( N(N(u)) \) are the second order neighbors of \( u \) in \( G \) excluding \( u \), and \( c_{uv} \) is the pairwise clustering coefficient between nodes \( u \) and \( v \).

The mode selects the function for \( c_{uv} \) which can be:

**dot:**

\[ c_{uv} = \frac{|N(u) \cap N(v)|}{|N(u) \cup N(v)|} \]

**min:**

\[ c_{uv} = \frac{|N(u) \cap N(v)|}{\min(|N(u)|, |N(v)|)} \]

**max:**

\[ c_{uv} = \frac{|N(u) \cap N(v)|}{\max(|N(u)|, |N(v)|)} \]

**Parameters**

\( G \) : graph

A bipartite graph

**nodes** : list or iterable (optional)

Compute bipartite clustering for these nodes. The default is all nodes in \( G \).

**mode** : string

The pairwise bipartite clustering method to be used in the computation. It must be “dot”, “max”, or “min”.

**Returns**

clustering : dictionary

A dictionary keyed by node with the clustering coefficient value.

**See Also:**

robins_alexander_clustering, square_clustering, average_clustering

**References**

[R159]

**Examples**

```python
>>> from networkx.algorithms import bipartite
>>> G = nx.path_graph(4)  # path graphs are bipartite
>>> c = bipartite.clustering(G)
>>> c[0]
0.5
>>> c = bipartite.clustering(G, mode='min')
>>> c[0]
1.0
```
robins_alexander_clustering

robins_alexander_clustering(G)

Compute the bipartite clustering of G.

Robins and Alexander [R160] defined bipartite clustering coefficient as four times the number of four cycles $C_4$ divided by the number of three paths $L_3$ in a bipartite graph:

$$CC_4 = \frac{4 \cdot C_4}{L_3}$$

Parameters

- **G**: graph
  - A bipartite graph

Returns

- **clustering**: float
  - The Robins and Alexander bipartite clustering for the input graph.

See Also:

- `latapy_clustering`, `square_clustering`

References

[R160]

Examples

```python
>>> from networkx.algorithms import bipartite
>>> G = nx.davis_southern_women_graph()
>>> print(round(bipartite.robins_alexander_clustering(G), 3))
0.468
```

4.3.5 Redundancy

Node redundancy for bipartite graphs.

```python
node_redundancy(G, nodes)
```

Compute bipartite node redundancy coefficient.

node_redundancy

node_redundancy(G, nodes=None)

Compute bipartite node redundancy coefficient.

The redundancy coefficient of a node $v$ is the fraction of pairs of neighbors of $v$ that are both linked to other nodes. In a one-mode projection these nodes would be linked together even if $v$ were not there.

$$rc(v) = \frac{|\{(u, w) \subseteq N(v), \exists v' \neq v, (v', u) \in E \text{ and } (v', w) \in E\}|}{\frac{1}{2}|N(v)||N(v)|-1}$$

where $N(v)$ are the neighbors of $v$ in $G$.

Parameters

- **G**: graph
  - A bipartite graph
nodes : list or iterable (optional)
    Compute redundancy for these nodes. The default is all nodes in G.

Returns redundancy : dictionary
    A dictionary keyed by node with the node redundancy value.

References

[R164]

Examples

```python
>>> from networkx.algorithms import bipartite
>>> G = nx.cycle_graph(4)
>>> rc = bipartite.node_redundancy(G)
>>> rc[0]
1.0

Compute the average redundancy for the graph:
```n
```python
>>> sum(rc.values()) / len(G)
1.0

Compute the average redundancy for a set of nodes:
```n
```python
>>> nodes = [0, 2]
>>> sum(rc[n] for n in nodes) / len(nodes)
1.0

4.3.6 Centrality

closeness centrality

closeness_centrality(G, nodes[, normalized])  Compute the closeness centrality for nodes in a bipartite network.

degree centrality

degree_centrality(G, nodes)  Compute the degree centrality for nodes in a bipartite network.

betweenness centrality

betweenness_centrality(G, nodes)  Compute betweenness centrality for nodes in a bipartite network.

closeness centrality

closeness_centrality(G, nodes, normalized=True)
    Compute the closeness centrality for nodes in a bipartite network.

    The closeness of a node is the distance to all other nodes in the graph or in the case that the graph is not connected
to all other nodes in the connected component containing that node.

    Parameters
    G : graph
        A bipartite network

    nodes : list or container
        Container with all nodes in one bipartite node set.

    normalized : bool, optional
        If True (default) normalize by connected component size.
Returns `closeness`: dictionary

Dictionary keyed by node with bipartite closeness centrality as the value.

See Also:

`betweenness_centrality, degree_centrality, sets, is_bipartite`

Notes

The nodes input parameter must contain all nodes in one bipartite node set, but the dictionary returned contains all nodes from both node sets.

Closeness centrality is normalized by the minimum distance possible. In the bipartite case the minimum distance for a node in one bipartite node set is 1 from all nodes in the other node set and 2 from all other nodes in its own set [R155]. Thus the closeness centrality for node \( v \) in the two bipartite sets \( U \) with \( n \) nodes and \( V \) with \( m \) nodes is

\[
c_v^U = \frac{m + 2(n - 1)}{d}, \text{ for } v \in U,
\]

\[
c_v^V = \frac{n + 2(m - 1)}{d}, \text{ for } v \in V,
\]

where \( d \) is the sum of the distances from \( v \) to all other nodes.

Higher values of closeness indicate higher centrality.

As in the unipartite case, setting normalized=True causes the values to normalized further to \( n-1 / \text{size}(G) - 1 \) where \( n \) is the number of nodes in the connected part of graph containing the node. If the graph is not completely connected, this algorithm computes the closeness centrality for each connected part separately.

References

[R155]

degree_centrality

degree_centrality \((G, \text{nodes})\)

Compute the degree centrality for nodes in a bipartite network.

The degree centrality for a node \( v \) is the fraction of nodes connected to it.

Parameters

- **G**: graph
  
  A bipartite network

- **nodes**: list or container
  
  Container with all nodes in one bipartite node set.

Returns `centrality`: dictionary

Dictionary keyed by node with bipartite degree centrality as the value.

See Also:

`betweenness_centrality, closeness_centrality, sets, is_bipartite`
Notes

The nodes input parameter must contain all nodes in one bipartite node set, but the dictionary returned contains all nodes from both bipartite node sets.

For unipartite networks, the degree centrality values are normalized by dividing by the maximum possible degree (which is \( n - 1 \) where \( n \) is the number of nodes in \( G \)).

In the bipartite case, the maximum possible degree of a node in a bipartite node set is the number of nodes in the opposite node set \([R156]\). The degree centrality for a node \( v \) in the bipartite sets \( U \) with \( n \) nodes and \( V \) with \( m \) nodes is

\[
d_v = \frac{\text{deg}(v)}{m}, \text{for } v \in U,
\]

\[
d_v = \frac{\text{deg}(v)}{n}, \text{for } v \in V,
\]

where \( \text{deg}(v) \) is the degree of node \( v \).

References

[R156]

betweenness_centrality

betweenness_centrality \((G, \text{nodes})\)

Compute betweenness centrality for nodes in a bipartite network.

Betweenness centrality of a node \( v \) is the sum of the fraction of all-pairs shortest paths that pass through \( v \).

Values of betweenness are normalized by the maximum possible value which for bipartite graphs is limited by the relative size of the two node sets \([R154]\).

Let \( n \) be the number of nodes in the node set \( U \) and \( m \) be the number of nodes in the node set \( V \), then nodes in \( U \) are normalized by dividing by

\[
\frac{1}{2} \left[ m^2(s+1)^2 + m(s+1)(2t-s-1) - t(2s-t+3) \right],
\]

where

\[
s = (n-1) \div m, t = (n-1) \mod m,
\]

and nodes in \( V \) are normalized by dividing by

\[
\frac{1}{2} \left[ n^2(p+1)^2 + n(p+1)(2r-p-1) - r(2p-r+3) \right],
\]

where,

\[
p = (m-1) \div n, r = (m-1) \mod n.
\]

Parameters

\( G \) : graph

A bipartite graph

\( \text{nodes} \) : list or container

Container with all nodes in one bipartite node set.
Returns  

betweenness : dictionary

Dictionary keyed by node with bipartite betweenness centrality as the value.

See Also:

degree_centrality, closeness_centrality, sets, is_bipartite

Notes

The nodes input parameter must contain all nodes in one bipartite node set, but the dictionary returned contains all nodes from both node sets.

References

[R154]

4.4 Blockmodeling

Functions for creating network blockmodels from node partitions.

Created by Drew Conway <drew.conway@nyu.edu> Copyright (c) 2010. All rights reserved.

blockmodel(G, partitions[, multigraph])  Returns a reduced graph constructed using the generalized block modeling technique.

4.4.1 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

[R166]

Examples

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

4.5 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.

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

4.5.1 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.
4.5.2 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.

4.6 Centrality

4.6.1 Degree

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>degree_centrality(G)</td>
<td>Compute the degree centrality for nodes.</td>
</tr>
<tr>
<td>in_degree_centrality(G)</td>
<td>Compute the in-degree centrality for nodes.</td>
</tr>
<tr>
<td>out_degree_centrality(G)</td>
<td>Compute the out-degree centrality for nodes.</td>
</tr>
</tbody>
</table>

**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.

in_degree_centrality

\textit{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.

\textbf{Parameters} \hspace{1em} G : graph

A NetworkX graph

\textbf{Returns} \hspace{1em} nodes : dictionary

Dictionary of nodes with in-degree centrality as values.

\textbf{See Also:}

degree_centrality, out_degree_centrality

Notes

The degree centrality values are normalized by dividing by the maximum possible degree in a simple graph n-1 where n is the number of nodes in G.

For multigraphs or graphs with self loops the maximum degree might be higher than n-1 and values of degree centrality greater than 1 are possible.

out_degree_centrality

\textit{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.

\textbf{Parameters} \hspace{1em} G : graph

A NetworkX graph

\textbf{Returns} \hspace{1em} nodes : dictionary

Dictionary of nodes with out-degree centrality as values.

\textbf{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.

4.6.2 Closeness

closeness_centrality

closeness_centrality(G, u=None, distance=None, normalized=True) Compute closeness centrality for nodes.

Closeness centrality [R171] of a node u is the reciprocal of the sum of the shortest path distances from u to all n−1 other nodes. Since the sum of distances depends on the number of nodes in the graph, closeness is normalized by the sum of minimum possible distances n−1.

\[ C(u) = \frac{n - 1}{\sum_{v=1}^{n-1} d(v, u)}, \]

where \( d(v, u) \) is the shortest-path distance between \( v \) and \( u \), and \( n \) is the number of nodes in the graph.

Notice that higher values of closeness indicate higher centrality.

Parameters

- **G**: graph
  A NetworkX graph
- **u**: node, optional
  Return only the value for node u
- **distance**: edge attribute key, optional (default=None)
  Use the specified edge attribute as the edge distance in shortest path calculations
- **normalized**: bool, optional
  If True (default) normalize by the number of nodes in the connected part of the graph.

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 \((n - 1)/(|G| - 1)\) 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.
If the ‘distance’ keyword is set to an edge attribute key then the shortest-path length will be computed using Dijkstra’s algorithm with that edge attribute as the edge weight.

References

[R171]

### 4.6.3 Betweenness

#### betweenness_centrality(G[, k, normalized, ...])
Compute the shortest-path betweenness centrality for nodes.

#### edge_betweenness_centrality(G[, normalized, ...])
Compute betweenness centrality for edges.

**betweenness_centrality**

Compute the shortest-path betweenness centrality for nodes.

Betweenness centrality of a node $v$ is the sum of the fraction of all-pairs shortest paths that pass through $v$:

$$ c_B(v) = \sum_{s,t \in V} \frac{\sigma(s, t|v)}{\sigma(s, t)} $$

where $V$ is the set of nodes, $\sigma(s, t)$ is the number of shortest $(s, t)$-paths, and $\sigma(s, t|v)$ is the number of those paths passing through some node $v$ other than $s, t$. If $s = t$, $\sigma(s, t) = 1$, and if $v \in s, t$, $\sigma(s, t|v) = 0$ [R169].

**Parameters**

- **G**: graph
  - A NetworkX graph
- **k**: int, optional (default=None)
  - If $k$ is not None use $k$ node samples to estimate betweenness. The value of $k \leq n$ where $n$ is the number of nodes in the graph. Higher values give better approximation.
- **normalized**: bool, optional
  - If True the betweenness values are normalized by $2/((n - 1)(n - 2))$ for graphs, and $1/((n - 1)(n - 2))$ for directed graphs where $n$ is the number of nodes in $G$.
- **weight**: None or string, optional
  - If None, all edge weights are considered equal. Otherwise holds the name of the edge attribute used as weight.
- **endpoints**: bool, optional
  - If True include the endpoints in the shortest path counts.

**Returns**

- **nodes**: dictionary
  - Dictionary of nodes with betweenness centrality as the value.

**See Also**:

- `edge_betweenness_centrality`, `load_centrality`
Notes

The algorithm is from Ulrik Brandes [R168]. See [R169] for details on algorithms for variations and related metrics.

For approximate betweenness calculations set $k=\#\text{samples}$ to use $k$ nodes (“pivots”) to estimate the betweenness values. For an estimate of the number of pivots needed see [R170].

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

[R168], [R169], [R170]

edge_betweenness_centrality

**edge_betweenness_centrality** ($G$, normalized=True, weight=None)

Compute betweenness centrality for edges.

Betweenness centrality of an edge $e$ is the sum of the fraction of all-pairs shortest paths that pass through $e$:

$$c_B(v) = \sum_{s,t \in V} \frac{\sigma(s,t|e)}{\sigma(s,t)}$$

where $V$ is the set of nodes, `$\sigma(s, t)$` is the number of shortest $(s, t)$-paths, and $\sigma(s, t|e)$ is the number of those paths passing through edge $e$ [R186].

Parameters

- **G** : graph
  A NetworkX graph

- **normalized** : bool, optional
  If True the betweenness values are normalized by $2/(n(n-1))$ for graphs, and $1/(n(n-1))$ for directed graphs where $n$ is the number of nodes in $G$. 

- **weight** : None or string, optional
  If None, all edge weights are considered equal. Otherwise holds the name of the edge attribute used as weight.

Returns

- **edges** : dictionary
  Dictionary of edges with betweenness centrality as the value.

See Also:

`betweenness_centrality`, `edge_load`

Notes

The algorithm is from Ulrik Brandes [R185].

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.
4.6.4 Current Flow Closeness

current_flow_closeness_centrality(G[, ...])  Compute current-flow closeness centrality for nodes.

current_flow_closeness_centrality

current_flow_closeness_centrality(G, weight='weight', dtype=<type 'float'>, solver='lu')
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
- **dtype**: data type (float)
  Default data type for internal matrices. Set to np.float32 for lower memory consumption.
- **solver**: string (default='lu')
  Type of linear solver to use for computing the flow matrix. Options are “full” (uses most memory), “lu” (recommended), and “cg” (uses least memory).

Returns
- **nodes**: dictionary
  Dictionary of nodes with current flow closeness centrality as the value.

See Also:
- closeness_centrality

Notes

The algorithm is from Brandes [R182].

See also [R183] for the original definition of information centrality.

References

[R182], [R183]

4.6.5 Current-Flow Betweenness

current_flow_betweenness_centrality(G[, ...])  Compute current-flow betweenness centrality for nodes.

edge_current_flow_betweenness_centrality(G)  Compute current-flow betweenness centrality for edges.

approximate_current_flow_betweenness_centrality(G)  Compute the approximate current-flow betweenness centrality for nodes.
current_flow_betweenness_centrality

**current_flow_betweenness_centrality** *(G, normalized=True, weight='weight', dtype='float', solver='full')*

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 [R181].

**Parameters**

- **G** : graph  
  A NetworkX graph

- **normalized** : bool, optional (default=True)
  If True the betweenness values are normalized by \(2/(n-1)(n-2)\) where \(n\) is the number of nodes in \(G\).

- **weight** : string or None, optional (default='weight')
  Key for edge data used as the edge weight. If None, then use 1 as each edge weight.

- **dtype** : data type (float)
  Default data type for internal matrices. Set to np.float32 for lower memory consumption.

- **solver** : string (default='lu')
  Type of linear solver to use for computing the flow matrix. Options are “full” (uses most memory), “lu” (recommended), and “cg” (uses least memory).

**Returns**

- **nodes** : dictionary
  Dictionary of nodes with betweenness centrality as the value.

**See Also:**

- approximate_current_flow_betweenness_centrality, betweenness_centrality, edge_betweenness_centrality, edge_current_flow_betweenness_centrality

**Notes**

Current-flow betweenness can be computed in \(O(I(n-1) + mn \log n)\) time [R180], where \(I(n-1)\) is the time needed to compute the inverse Laplacian. For a full matrix this is \(O(n^3)\) but using sparse methods you can achieve \(O(nm \sqrt{k})\) where \(k\) is the Laplacian matrix condition number.

The space required is \(O(nw)\) where \(w\) is the width of the sparse Laplacian matrix. Worse case is \(w = n\) for \(O(n^2)\).

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

**References**

[R180], [R181]
edge_current_flow_betweenness_centrality

edge_current_flow_betweenness_centrality(G, normalized=True, weight='weight', dtype=<type 'float'>, solver='full')

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 [R188].

**Parameters**

- **G**: graph
  A NetworkX graph

- **normalized**: bool, optional (default=True)
  If True the betweenness values are normalized by $2/(n-1)(n-2)$ where $n$ is the number of nodes in $G$.

- **weight**: string or None, optional (default='weight')
  Key for edge data used as the edge weight. If None, then use 1 as each edge weight.

- **dtype**: data type (float)
  Default data type for internal matrices. Set to np.float32 for lower memory consumption.

- **solver**: string (default='lu')
  Type of linear solver to use for computing the flow matrix. Options are “full” (uses most memory), “lu” (recommended), and “cg” (uses least memory).

**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**

Current-flow betweenness can be computed in $O(I(n-1) + mn \log n)$ time [R187], where $I(n-1)$ is the time needed to compute the inverse Laplacian. For a full matrix this is $O(n^3)$ but using sparse methods you can achieve $O(nm\sqrt{k})$ where $k$ is the Laplacian matrix condition number.

The space required is $O(nw)$ where $w$ is the width of the sparse Laplacian matrix. Worse case is $w = n$ for $O(n^2)$.

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

**References**

[R187], [R188]
approximate_current_flow_betweenness_centrality

approximate_current_flow_betweenness_centrality \( (G, \text{normalized}=\text{True}, \text{weight}=‘\text{weight’},\text{dtype}=\text{<type ‘float’>}, \text{solver}=‘\text{full’}, \epsilon=0.5, \text{kmax}=10000) \)

Compute the approximate current-flow betweenness centrality for nodes.
Approximates the current-flow betweenness centrality within absolute error of \( \epsilon \) with high probability [R167].

Parameters

\( G \): graph
A NetworkX graph

\text{normalized} : \text{bool, optional (default=True)}
If True the betweenness values are normalized by \( 2/(n-1)(n-2) \) where \( n \) is the number of nodes in \( G \).

\text{weight} : \text{string or None, optional (default=’weight’)}
Key for edge data used as the edge weight. If None, then use 1 as each edge weight.

\text{dtype} : \text{data type (float)}
Default data type for internal matrices. Set to np.float32 for lower memory consumption.

\text{solver} : \text{string (default=’lu’)}
Type of linear solver to use for computing the flow matrix. Options are “full” (uses most memory), “lu” (recommended), and “cg” (uses least memory).

\text{epsilon} : \text{float}
Absolute error tolerance.

\text{kmax} : \text{int}
Maximum number of sample node pairs to use for approximation.

Returns \( \text{nodes} \) : dictionary
Dictionary of nodes with betweenness centrality as the value.

See Also:
current_flow_betweenness_centrality

Notes

The running time is \( O((1/\epsilon^2)m\sqrt{k} \log n) \) and the space required is \( O(m) \) for \( n \) nodes and \( m \) edges.
If the edges have a ‘weight’ attribute they will be used as weights in this algorithm. Unspecified weights are set to 1.

References

[R167]
4.6.6 Eigenvector
eigenvector_centrality

**eigenvector_centrality** (G[, max_iter, tol, ...])  
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**: integer, 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.
- **weight**: None or string, optional  
  If None, all edge weights are considered equal. Otherwise holds the name of the edge attribute used as weight.

**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 “left” eigenvector centrality which corresponds to the in-edges in the graph. For out-edges eigenvector centrality first reverse the graph with G.reverse().

**Examples**

```python
>>> 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']
```
**eigenvector_centrality_numpy**

*eigenvector_centrality_numpy* *(G, weight='weight')*

Compute the eigenvector centrality for the graph G.

**Parameters**

- **G**: graph
  - A networkx graph

- **weight**: None or string, optional
  - The name of the edge attribute used as weight. If None, all edge weights are considered equal.

**Returns**

- **nodes**: dictionary
  - Dictionary of nodes with eigenvector centrality as the value.

**See Also:**

eigenvector_centrality, pagerank, hits

**Notes**

This algorithm uses the SciPy sparse eigenvalue solver (ARPACK) to find the largest eigenvalue/eigenvector pair.

For directed graphs this is “left” eigenvector centrality which corresponds to the in-edges in the graph. For out-edges eigenvector centrality first reverse the graph with G.reverse().

**Examples**

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

**katz_centrality**

*katz_centrality* *(G, alpha=0.1, beta=1.0, max_iter=1000, tol=1e-06, nstart=None, normalized=True, weight='weight')*

Compute the Katz centrality for the nodes of the graph G.

Katz centrality is related to eigenvalue centrality and PageRank. The Katz centrality for node *i* is

\[ x_i = \alpha \sum_j A_{ij} x_j + \beta, \]

where \( A \) is the adjacency matrix of the graph G with eigenvalues \( \lambda \).

The parameter \( \beta \) controls the initial centrality and

\[ \alpha < \frac{1}{\lambda_{max}}. \]

Katz centrality computes the relative influence of a node within a network by measuring the number of the immediate neighbors (first degree nodes) and also all other nodes in the network that connect to the node under consideration through these immediate neighbors.
Extra weight can be provided to immediate neighbors through the parameter $\beta$. Connections made with distant neighbors are, however, penalized by an attenuation factor $\alpha$ which should be strictly less than the inverse largest eigenvalue of the adjacency matrix in order for the Katz centrality to be computed correctly. More information is provided in [R190] .

**Parameters**

- **G**: graph
  - A NetworkX graph
- **alpha**: float
  - Attenuation factor
- **beta**: scalar or dictionary, optional (default=1.0)
  - Weight attributed to the immediate neighborhood. If not a scalar the dictionary must have an value for every node.
- **max_iter**: integer, optional (default=1000)
  - Maximum number of iterations in power method.
- **tol**: float, optional (default=1.0e-6)
  - Error tolerance used to check convergence in power method iteration.
- **nstart**: dictionary, optional
  - Starting value of Katz iteration for each node.
- **normalized**: bool, optional (default=True)
  - If True normalize the resulting values.
- **weight**: None or string, optional
  - If None, all edge weights are considered equal. Otherwise holds the name of the edge attribute used as weight.

**Returns**

- **nodes**: dictionary
  - Dictionary of nodes with Katz centrality as the value.

**See Also:**

- katz_centrality_numpy, eigenvector_centrality, eigenvector_centrality_numpy, pagerank, hits

**Notes**

This algorithm it uses the power method to find the eigenvector corresponding to the largest eigenvalue of the adjacency matrix of $G$. The constant alpha should be strictly less than the inverse of largest eigenvalue of the adjacency matrix for the algorithm to converge. The iteration will stop after max_iter iterations or an error tolerance of number_of_nodes($G$)*tol has been reached.

When $\alpha = 1/\lambda_{\text{max}}$ and $\beta = 1$ Katz centrality is the same as eigenvector centrality.

For directed graphs this finds “left” eigenvectors which corresponds to the in-edges in the graph. For out-edges Katz centrality first reverse the graph with $G$.reverse().

**References**

[R190]

4.6. Centrality 175
Examples

```python
>>> import math
>>> G = nx.path_graph(4)
>>> phi = (1+math.sqrt(5))/2.0 # largest eigenvalue of adj matrix
>>> centrality = nx.katz_centrality(G,1/phi-0.01)
>>> for n,c in sorted(centrality.items()):
...     print("%d %0.2f"%(n,c))
0 0.37
1 0.60
2 0.60
3 0.37
```

**katz_centrality_numpy**

The `katz_centrality_numpy` function computes the Katz centrality for the graph `G`.

Katz centrality is related to eigenvector centrality and PageRank. The Katz centrality for node \( i \) is

\[
x_i = \alpha \sum_j A_{ij} x_j + \beta,
\]

where \( A \) is the adjacency matrix of the graph \( G \) with eigenvalues \( \lambda \).

The parameter \( \beta \) controls the initial centrality and

\[
\alpha < \frac{1}{\lambda_{\text{max}}}
\]

Katz centrality computes the relative influence of a node within a network by measuring the number of the immediate neighbors (first degree nodes) and also all other nodes in the network that connect to the node under consideration through these immediate neighbors.

Extra weight can be provided to immediate neighbors through the parameter \( \beta \). Connections made with distant neighbors are, however, penalized by an attenuation factor \( \alpha \) which should be strictly less than the inverse largest eigenvalue of the adjacency matrix in order for the Katz centrality to be computed correctly. More information is provided in [R191].

**Parameters**

- **G**: graph
  - A NetworkX graph
- **alpha**: float
  - Attenuation factor
- **beta**: scalar or dictionary, optional (default=1.0)
  - Weight attributed to the immediate neighborhood. If not a scalar the dictionary must have an value for every node.
- **normalized**: bool
  - If True normalize the resulting values.
- **weight**: None or string, optional
  - If None, all edge weights are considered equal. Otherwise holds the name of the edge attribute used as weight.

**Returns**

- **nodes**: dictionary
Dictionary of nodes with Katz centrality as the value.

See Also:

katz_centrality, eigenvector_centrality_numpy, eigenvector_centrality, pagerank, hits

Notes

This algorithm uses a direct linear solver to solve the above equation. The constant alpha should be strictly less
than the inverse of largest eigenvalue of the adjacency matrix for there to be a solution. When \( \alpha = 1/\lambda_{\text{max}} \) and
\( \beta = 1 \) Katz centrality is the same as eigenvector centrality.

For directed graphs this finds “left” eigenvectors which corresponds to the in-edges in the graph. For out-edges
Katz centrality first reverse the graph with G.reverse().

References

[R191]

Examples

```python
>>> import math
>>> G = nx.path_graph(4)
>>> phi = (1+math.sqrt(5))/2.0 # largest eigenvalue of adj matrix
>>> centrality = nx.katz_centrality_numpy(G,1/phi)
>>> for n,c in sorted(centrality.items()):
...     print("%d %0.2f"%(n,c))
0 0.37
1 0.60
2 0.60
3 0.37
```

4.6.7 Communicability

<table>
<thead>
<tr>
<th>communicability(G)</th>
<th>Return communicability between all pairs of nodes in G.</th>
</tr>
</thead>
<tbody>
<tr>
<td>communicability_exp(G)</td>
<td>Return communicability between all pairs of nodes in G.</td>
</tr>
<tr>
<td>communicability_centrality(G)</td>
<td>Return communicability centrality for each node in G.</td>
</tr>
<tr>
<td>communicability_centrality_exp(G)</td>
<td>Return the communicability centrality for each node of G</td>
</tr>
<tr>
<td>communicability_betweenness_centrality(G[, ...])</td>
<td>Return communicability betweenness for all pairs of nodes in G.</td>
</tr>
<tr>
<td>estrada_index(G)</td>
<td>Return the Estrada index of a the graph G.</td>
</tr>
</tbody>
</table>

communicability

`communicability(G)`

Return communicability between all pairs of nodes in G.

The communicability between pairs of nodes in G is the sum of closed walks of different lengths starting at node u and ending at node v.

Parameters

G: graph

4.6. Centrality 177
Returns  
comm: dictionary of dictionaries

Dictionary of dictionaries keyed by nodes with communicability as the value.

Raises  
NetworkXError

If the graph is not undirected and simple.

See Also:

communicability_centrality_exp  Communicability centrality for each node of G using matrix exponential.

communicability_centrality  Communicability centrality for each node in G using spectral decomposition.

communicability  Communicability between pairs of nodes in G.

Notes

This algorithm uses a spectral decomposition of the adjacency matrix. Let G=(V,E) be a simple undirected graph. Using the connection between the powers of the adjacency matrix and the number of walks in the graph, the communicability between nodes u and v based on the graph spectrum is [R172]

\[ C(u, v) = \sum_{j=1}^{n} \phi_j(u) \phi_j(v) e^{\lambda_j}, \]

where \( \phi_j(u) \) is the \( u \)th element of the \( j \)th orthonormal eigenvector of the adjacency matrix associated with the eigenvalue \( \lambda_j \).

References

[R172]

Examples

```python
>>> G = nx.Graph([(0,1),(1,2),(1,5),(5,4),(2,4),(2,3),(4,3),(3,6)])
>>> c = nx.communicability(G)
```

communicability_exp

communicability_exp(G)

Return communicability between all pairs of nodes in G.

Communicability between pair of node (u,v) of node in G is the sum of closed walks of different lengths starting at node u and ending at node v.

Parameters  
G: graph

Returns  
comm: dictionary of dictionaries

Dictionary of dictionaries keyed by nodes with communicability as the value.

Raises  
NetworkXError
If the graph is not undirected and simple.

See Also:

- **communicability_centrality_exp** Communicability centrality for each node of G using matrix exponential.
- **communicability_centrality** Communicability centrality for each node in G using spectral decomposition.
- **communicability_exp** Communicability between all pairs of nodes in G using spectral decomposition.

Notes

This algorithm uses matrix exponentiation of the adjacency matrix.

Let \(G=(V,E)\) be a simple undirected graph. Using the connection between the powers of the adjacency matrix and the number of walks in the graph, the communicability between nodes \(u\) and \(v\) is \([R179]\),

\[
C(u, v) = (e^A)_{uv},
\]

where \(A\) is the adjacency matrix of \(G\).

References

[R179]

Examples

```python
>>> G = nx.Graph([(0, 1), (1, 2), (1, 5), (5, 4), (2, 4), (2, 3), (4, 3), (3, 6)])
>>> c = nx.communicability_exp(G)
```

**communicability_centrality**

**communicability_centrality** \((G)\)

Return communicability centrality for each node in \(G\).

Communicability centrality, also called subgraph centrality, of a node \(n\) is the sum of closed walks of all lengths starting and ending at node \(n\).

Parameters

- **G**: graph

Returns

- nodes: dictionary
  Dictionary of nodes with communicability centrality as the value.

Raises

- NetworkXError
  If the graph is not undirected and simple.

See Also:

- **communicability** Communicability between all pairs of nodes in \(G\).
- **communicability_centrality** Communicability centrality for each node of \(G\).
Notes

This version of the algorithm computes eigenvalues and eigenvectors of the adjacency matrix.

Communicability centrality of a node \( u \) in \( G \) can be found using a spectral decomposition of the adjacency matrix [R175] [R176],

\[
SC(u) = \sum_{j=1}^{N} (v_j^u)^2 e^{\lambda_j},
\]

where \( v_j \) is an eigenvector of the adjacency matrix \( A \) of \( G \) corresponding to the eigenvalue \( \lambda_j \).

References

[R175], [R176]

Examples

```python
>>> G = nx.Graph([(0,1),(1,2),(1,5),(5,4),(2,4),(2,3),(4,3),(3,6)])
>>> sc = nx.communicability_centrality(G)
```

**communicability_centrality_exp**

**communicability_centrality_exp** \((G)\)

Return the communicability centrality for each node of \( G \).

Communicability centrality, also called subgraph centrality, of a node \( n \) is the sum of closed walks of all lengths starting and ending at node \( n \).

**Parameters**  
\( G \): graph

**Returns**  
nodes:dictionary

Dictionary of nodes with communicability centrality as the value.

**Raises**  
NetworkXError

If the graph is not undirected and simple.

See Also:

**communicability** Communicability between all pairs of nodes in \( G \).

**communicability_centrality** Communicability centrality for each node of \( G \).

Notes

This version of the algorithm exponentiates the adjacency matrix. The communicability centrality of a node \( u \) in \( G \) can be found using the matrix exponential of the adjacency matrix of \( G \) [R177] [R178],

\[
SC(u) = (e^A)_{uu}.
\]
References

[R177], [R178]

Examples

```python
>>> G = nx.Graph([(0,1), (1, 2), (1, 5), (5, 4), (2, 4), (2, 3), (4, 3), (3, 6)])
>>> sc = nx.communicability_centrality_exp(G)
```

**communicability_betweenness_centrality**

`communicability_betweenness_centrality(G, normalized=True)`

Return communicability betweenness for all pairs of nodes in G.

**Parameters**

| G       | graph |

**Returns**

| nodes:dictionary |

Dictionary of nodes with communicability betweenness as the value.

**Raises**

| NetworkXError |

If the graph is not undirected and simple.

**See Also:**

- `communicability` Communicability between all pairs of nodes in G.
- `communicability_centrality` Communicability centrality for each node of G using matrix exponential.
- `communicability_centrality_exp` Communicability centrality for each node in G using spectral decomposition.

**Notes**

Let $G = (V, E)$ be a simple undirected graph with $n$ nodes and $m$ edges, and $A$ denote the adjacency matrix of $G$.

Let $G(r) = (V, E(r))$ be the graph resulting from removing all edges connected to node $r$ but not the node itself.

The adjacency matrix for $G(r)$ is $A + E(r)$, where $E(r)$ has nonzeros only in row and column $r$.

The communicability betweenness of a node $r$ is [R174]

$$
\omega_r = \frac{1}{C} \sum_p \sum_q G^p_{rq}(G^p_{pq} - (e^A + E(r))_{pq}), p \neq q, q \neq r,
$$

where $G^p_{pq} = (e^A - (e^A + E(r))_{pq})$ is the number of walks involving node $r$, $G^p_{pq} = (e^A)_{pq}$ is the number of closed walks starting at node $p$ and ending at node $q$, and $C = (n - 1)^2 - (n - 1)$ is a normalization factor equal to the number of terms in the sum.

4.6. Centrality 181
The resulting $\omega_r$ takes values between zero and one. The lower bound cannot be attained for a connected graph, and the upper bound is attained in the star graph.

**References**

[R174]

**Examples**

```python
g = nx.Graph([(0, 1), (1, 2), (1, 5), (5, 4), (2, 4), (2, 3), (4, 3), (3, 6)])
cbc = nx.communicability_betweenness_centrality(g)
```

**estra_index**

`estra_index(G)`  
Return the Estrada index of the graph $G$.

**Parameters**  
$G$: graph

**Returns**  
estra index: float

**Raises**  
NetworkXError  
If the graph is not undirected and simple.

**See Also:**

`estra_index_exp`

**Notes**

Let $G = (V, E)$ be a simple undirected graph with $n$ nodes and let $\lambda_1 \leq \lambda_2 \leq \cdots \lambda_n$ be a non-increasing ordering of the eigenvalues of its adjacency matrix $A$. The Estrada index is

$$EE(G) = \sum_{j=1}^{n} e^{\lambda_j}.$$

**References**

[R189]

**Examples**

```python
g = nx.Graph([(0, 1), (1, 2), (1, 5), (5, 4), (2, 4), (2, 3), (4, 3), (3, 6)])
ei = nx.estra_index(g)
```

### 4.6.8 Load

182 Chapter 4. Algorithms
load_centrality

**load_centrality** *(G, v=None, cutoff=None, normalized=True, weight=None)*

Compute load centrality for nodes.

The load centrality of a node is the fraction of all shortest paths that pass through that node.

**Parameters**

- **G** : graph
  A networkx graph

- **normalized** : bool, optional
  If True the betweenness values are normalized by \( b = \frac{b}{(n-1)(n-2)} \) where \( n \) is the number of nodes in \( G \).

- **weight** : None or string, optional
  If None, edge weights are ignored. Otherwise holds the name of the edge attribute used as weight.

- **cutoff** : bool, optional
  If specified, only consider paths of length \( \leq \) cutoff.

**Returns**

- **nodes** : dictionary
  Dictionary of nodes with centrality as the value.

**See Also**

- betweenness_centrality

**Notes**


edge_load

**edge_load** *(G, nodes=None, cutoff=False)*

Compute edge load.

**WARNING:**

This module is for demonstration and testing purposes.

### 4.6.9 Dispersion

**dispersion** *(G, u, v, normalized, alpha, b, c)*

Calculate dispersion between \( u \) and \( v \) in \( G \).
dispersion

**dispersion** \((G, u=None, v=None, normalized=True, alpha=1.0, b=0.0, c=0.0)\)

Calculate dispersion between \(u\) and \(v\) in \(G\).

A link between two actors \((u\) and \(v\)) has a high dispersion when their mutual ties \((s\) and \(t\)) are not well connected with each other.

**Parameters**

- **\(G\)** : graph
  A NetworkX graph.
- **\(u\)** : node, optional
  The source for the dispersion score (e.g. ego node of the network).
- **\(v\)** : node, optional
  The target of the dispersion score if specified.
- **normalized** : bool
  If True (default) normalize by the embededness of the nodes \((u\) and \(v\)).

**Returns**

- **\(\text{nodes}\)** : dictionary
  If \(u\) (\(v\)) is specified, returns a dictionary of nodes with dispersion score for all “target” (“source”) nodes. If neither \(u\) nor \(v\) is specified, returns a dictionary of dictionaries for all nodes ‘\(u\)’ in the graph with a dispersion score for each node ‘\(v\)’.

**Notes**

This implementation follows Lars Backstrom and Jon Kleinberg [R184]. Typical usage would be to run dispersion on the ego network \(G_u\) if \(u\) were specified. Running \(\text{dispersion()}\) with neither \(u\) nor \(v\) specified can take some time to complete.

**References**

[R184]

### 4.7 Chordal

Algorithms for chordal graphs.

A graph is chordal if every cycle of length at least 4 has a chord (an edge joining two nodes not adjacent in the cycle). [http://en.wikipedia.org/wiki/Chordal_graph](http://en.wikipedia.org/wiki/Chordal_graph)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>is_chordal(G)</code></td>
<td>Checks whether (G) is a chordal graph.</td>
</tr>
<tr>
<td><code>chordal_graph_cliques(G)</code></td>
<td>Returns the set of maximal cliques of a chordal graph.</td>
</tr>
<tr>
<td><code>chordal_graph_treewidth(G)</code></td>
<td>Returns the treewidth of the chordal graph (G).</td>
</tr>
<tr>
<td><code>find_induced_nodes(G, s, t[, treewidth_bound])</code></td>
<td>Returns the set of induced nodes in the path from (s) to (t).</td>
</tr>
</tbody>
</table>
4.7.1 is_chordal

**is_chordal** *(G)*
Checks whether G is a chordal graph.

A graph is chordal if every cycle of length at least 4 has a chord (an edge joining two nodes not adjacent in the cycle).

**Parameters**  
G : graph  
A NetworkX graph.

**Returns**  
chordal : bool  
True if G is a chordal graph and False otherwise.

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

**Notes**

The routine tries to go through every node following maximum cardinality search. It returns False when it finds that the separator for any node is not a clique. Based on the algorithms in [R194].

**References**

[R194]

**Examples**

```python
def is_chordal(G):
    # Implementation...
    return chordal
```

4.7.2 chordal_graph_cliques

**chordal_graph_cliques** *(G)*
Returns the set of maximal cliques of a chordal graph.

The algorithm breaks the graph in connected components and performs a maximum cardinality search in each component to get the cliques.

**Parameters**  
G : graph  
A NetworkX graph.

**Returns**  
cliques : A set containing the maximal cliques in G.

**Raises**  
NetworkXError
The algorithm does not support DiGraph, MultiGraph and MultiDiGraph. If the input graph is an instance of one of these classes, a NetworkXError is raised. The algorithm can only be applied to chordal graphs. If the input graph is found to be non-chordal, a NetworkXError is raised.

Examples

```python
>>> import networkx as nx
>>> e = [(1,2), (1,3), (2,3), (2,4), (3,4), (3,5), (3,6), (4,5), (4,6), (5,6), (7,8)]
>>> G = nx.Graph(e)
>>> G.add_node(9)
>>> setlist = nx.chordal_graph_cliques(G)
```

4.7.3 chordal_graph_treewidth

**chordal_graph_treewidth** *(G)*

Returns the treewidth of the chordal graph G.

**Parameters**

- **G**: graph
  
  A NetworkX graph

**Returns**

- **treewidth**: int
  
  The size of the largest clique in the graph minus one.

**Raises**

- **NetworkXError**
  
  The algorithm does not support DiGraph, MultiGraph and MultiDiGraph. If the input graph is an instance of one of these classes, a NetworkXError is raised. The algorithm can only be applied to chordal graphs. If the input graph is found to be non-chordal, a NetworkXError is raised.

References

[R192]

Examples

```python
>>> import networkx as nx
>>> e = [(1,2), (1,3), (2,3), (2,4), (3,4), (3,5), (3,6), (4,5), (4,6), (5,6), (7,8)]
>>> G = nx.Graph(e)
>>> G.add_node(9)
>>> nx.chordal_graph_treewidth(G)
3
```

4.7.4 find_induced_nodes

**find_induced_nodes** *(G, s, t, treewidth_bound=9223372036854775807)*

Returns the set of induced nodes in the path from s to t.

**Parameters**

- **G**: graph
  
  A chordal NetworkX graph
s : node
Source node to look for induced nodes

t : node
Destination node to look for induced nodes

treewidth_bound: float
Maximum treewidth acceptable for the graph H. The search for induced nodes will end as soon as the treewidth_bound is exceeded.

Returns I : Set of nodes
The set of induced nodes in the path from s to t in G

Raises NetworkXError
The algorithm does not support DiGraph, MultiGraph and MultiDiGraph. If the input graph is an instance of one of these classes, a NetworkXError is raised. The algorithm can only be applied to chordal graphs. If the input graph is found to be non-chordal, a NetworkXError is raised.

Notes
G must be a chordal graph and (s,t) an edge that is not in G.
If a treewidth_bound is provided, the search for induced nodes will end as soon as the treewidth_bound is exceeded.
The algorithm is inspired by Algorithm 4 in [R193]. A formal definition of induced node can also be found on that reference.

References

[R193]

Examples

>>> import networkx as nx
>>> G=nx.Graph()
>>> G = nx.generators.classic.path_graph(10)
>>> I = nx.find_induced_nodes(G,1,9,2)
>>> list(I)
[1, 2, 3, 4, 5, 6, 7, 8, 9]

4.8 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.

NetworkX Reference, Release 1.9

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>find_cliques(G)</td>
<td>Search for all maximal cliques in a graph.</td>
</tr>
<tr>
<td>make_max_clique_graph(G[, create_using, name])</td>
<td>Create the maximal clique graph of a graph.</td>
</tr>
<tr>
<td>make_clique_bipartite(G[, fpos, ...])</td>
<td>Create a bipartite clique graph from a graph G.</td>
</tr>
<tr>
<td>graph_clique_number(G[, cliques])</td>
<td>Return the clique number (size of the largest clique) for G.</td>
</tr>
<tr>
<td>graph_number_of_cliques(G[, cliques])</td>
<td>Returns the number of maximal cliques in G.</td>
</tr>
<tr>
<td>node_clique_number(G[, nodes, cliques])</td>
<td>Returns the size of the largest maximal clique containing each given node.</td>
</tr>
<tr>
<td>number_of_cliques(G[, nodes, cliques])</td>
<td>Returns the number of maximal cliques for each node.</td>
</tr>
<tr>
<td>cliques_containing_node(G[, nodes, cliques])</td>
<td>Returns a list of cliques containing the given node.</td>
</tr>
</tbody>
</table>

### 4.8.1 find_cliques

**find_cliques(G)**

Search for all maximal cliques in a graph.

Maximal cliques are the largest complete subgraph containing a given node. The largest maximal clique is sometimes called the maximum clique.

**Returns** generator of lists: genetor of member list for each maximal clique

**See Also:**

find_cliques_recursive, A

**Notes**

To obtain a list of cliques, use list(find_cliques(G)).

Based on the algorithm published by Bron & Kerbosch (1973) [R195] as adapated by Tomita, Tanaka and Takahashi (2006) [R196] and discussed in Cazals and Karande (2008) [R197]. The method essentially unrolls the recursion used in the references to avoid issues of recursion stack depth.

This algorithm is not suitable for directed graphs.

This algorithm ignores self-loops and parallel edges as clique is not conventionally defined with such edges.

There are often many cliques in graphs. This algorithm can run out of memory for large graphs.

**References**

[R195], [R196], [R197]

### 4.8.2 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.
4.8.3 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.

4.8.4 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.

4.8.5 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.

4.8.6 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.

4.8.7 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.

4.8.8 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.
4.9 Clustering

Algorithms to characterize the number of triangles in a graph.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>triangles(G[ nodes])</td>
<td>Compute the number of triangles.</td>
</tr>
<tr>
<td>transitivity(G)</td>
<td>Compute graph transitivity, the fraction of all possible triangles present.</td>
</tr>
<tr>
<td>clustering(G[ nodes, weight])</td>
<td>Compute the clustering coefficient for nodes.</td>
</tr>
<tr>
<td>average_clustering(G[ nodes, weight, ...])</td>
<td>Compute the average clustering coefficient for the graph G.</td>
</tr>
<tr>
<td>square_clustering(G[ nodes])</td>
<td>Compute the squares clustering coefficient for nodes.</td>
</tr>
</tbody>
</table>

4.9.1 triangles

`triangles(G, nodes=None)`

Compute the number of triangles.

Finds the number of triangles that include a node as one vertex.

**Parameters**

- `G` : graph
  A networkx graph
- `nodes` : container of nodes, optional (default= all nodes in G)
  Compute triangles for nodes in this container.

**Returns**

- `out` : dictionary
  Number of triangles keyed by node label.

**Notes**

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

**Examples**

```python
>>> 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]
```

4.9.2 transitivity

`transitivity(G)`

Compute graph transitivity, the fraction of all possible triangles present in G.

Possible triangles are identified by the number of “triads” (two edges with a shared vertex).

The transitivity is

\[ T = \frac{\#\text{triangles}}{\#\text{triads}}. \]
Parameters  
\( G \) : graph

Returns  
\( \text{out} \) : float

Transitivity

Examples

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

4.9.3 clustering

`clustering(G, nodes=None, weight=None)`

Compute the clustering coefficient for nodes.

For unweighted graphs, the clustering of a node \( u \) is the fraction of possible triangles through that node that exist,

\[
c_u = \frac{2T(u)}{\text{deg}(u)(\text{deg}(u) - 1)},
\]

where \( T(u) \) is the number of triangles through node \( u \) and \( \text{deg}(u) \) is the degree of \( u \).

For weighted graphs, the clustering is defined as the geometric average of the subgraph edge weights \[R200\],

\[
c_u = \frac{1}{\text{deg}(u)(\text{deg}(u) - 1))} \sum_{uv} (\hat{w}_{uv}\hat{w}_{uw}\hat{w}_{vw})^{1/3}.
\]

The edge weights \( \hat{w}_{uv} \) are normalized by the maximum weight in the network \( \hat{w}_{uv} = w_{uv}\max(w) \).

The value of \( c_u \) is assigned to 0 if \( \text{deg}(u) < 2 \).

Parameters  
\( G \) : graph

- **nodes** : container of nodes, optional (default=all nodes in \( G \))
  
  Compute clustering for nodes in this container.

- **weight** : string or None, optional (default=None)
  
  The edge attribute that holds the numerical value used as a weight. If None, then each edge has weight 1.

Returns  
\( \text{out} \) : float, or dictionary

- Clustering coefficient at specified nodes

Notes

Self loops are ignored.

References

\[R200\]
4.9.4 average_clustering

average_clustering(G, nodes=None, weight=None, count_zeros=True)

Compute the average clustering coefficient for the graph G.

The clustering coefficient for the 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
  - **nodes**: container of nodes, optional (default=all nodes in G)
  - **weight**: string or None, optional (default=None)
    - The edge attribute that holds the numerical value used as a weight. If None, then each edge has weight 1.
  - **count_zeros**: bool (default=False)
    - If False include only the nodes with nonzero clustering in the average.

Returns

- **avg**: float
  - Average clustering

Notes

This is a space saving routine; it might be faster to use the clustering function to get a list and then take the average.

Self loops are ignored.

References

[R198], [R199]

Examples

```python
>>> 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}
```
### 4.9.5 square_clustering

**square_clustering** *(G, nodes=None)*

Compute the squares clustering coefficient for nodes.

For each node return the fraction of possible squares that exist at the node [R201]

\[
C_4(v) = \frac{\sum_{u=1}^{k_v} \sum_{w=u+1}^{k_v} q_v(u, w)}{\sum_{u=1}^{k_v} \sum_{w=u+1}^{k_v} [a_v(u, w) + q_v(u, w)]},
\]

where \(q_v(u, w)\) are the number of common neighbors of \(u\) and \(w\) other than \(v\) (i.e., squares), and \(a_v(u, w) = (k_u - (1 + q_v(u, w) + \theta_{uw}))(k_w - (1 + q_v(u, w) + \theta_{uw}))\), where \(\theta_{uw} = 1\) if \(u\) and \(w\) are connected and 0 otherwise.

**Parameters**

- **G**: graph
- **nodes**: container of nodes, optional (default=all nodes in G)

  Compute clustering for nodes in this container.

**Returns**

- **c4**: dictionary

  A dictionary keyed by node with the square clustering coefficient value.

**Notes**

While \(C_3(v)\) (triangle clustering) gives the probability that two neighbors of node \(v\) are connected with each other, \(C_4(v)\) is the probability that two neighbors of node \(v\) share a common neighbor different from \(v\). This algorithm can be applied to both bipartite and unipartite networks.

**References**

[R201]

**Examples**

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

### 4.10 Communities

#### 4.10.1 K-Clique

**k_clique_communities** *(G, k[, cliques])*

Find k-clique communities in graph using the percolation method.
**k_clique_communities**

**k_clique_communities**(*G, k, cliques=None*)

Find k-clique communities in graph using the percolation method.

A k-clique community is the union of all cliques of size k that can be reached through adjacent (sharing k-1 nodes) k-cliques.

**Parameters**

- **G**: NetworkX graph
- **k**: int
  - Size of smallest clique
- **cliques**: list or generator
  - Precomputed cliques (use networkx.find_cliques(G))

**Returns**

Yields sets of nodes, one for each k-clique community.

**References**

[R202]

**Examples**

```python
g = nx.complete_graph(5)
k5 = nx.convert_node_labels_to_integers(g, first_label=2)
g.add_edges_from(k5.edges())
c = list(nx.k_clique_communities(g, 4))
list(c[0])
[0, 1, 2, 3, 4, 5, 6]
```

4.11 Components

4.11.1 Connectivity

Connected components.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>is_connected(G)</td>
<td>Return True if the graph is connected, false otherwise.</td>
</tr>
<tr>
<td>number_connected_components(G)</td>
<td>Return the number of connected components.</td>
</tr>
<tr>
<td>connected_components(G)</td>
<td>Generate connected components.</td>
</tr>
<tr>
<td>connected_component_subgraphs(G, copy)</td>
<td>Generate connected components as subgraphs.</td>
</tr>
<tr>
<td>node_connected_component(G, n)</td>
<td>Return the nodes in the component of graph containing node n.</td>
</tr>
</tbody>
</table>

**is_connected**

**is_connected**(*G*)

Return True if the graph is connected, false otherwise.

**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

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

number_connected_components

number_connected_components\( (G) \)

Return the number of connected components.

Parameters  
G : NetworkX graph

An undirected graph.

Returns  
n : integer

Number of connected components

See Also:
connected_components

Notes

For undirected graphs only.

connected_components

connected_components\( (G) \)

Generate connected components.

Parameters  
G : NetworkX graph

An undirected graph

Returns  
comp : generator of lists

A list of nodes for each component of G.

See Also:
strongly_connected_components
Notes

For undirected graphs only.

Examples

Generate a sorted list of connected components, largest first.

```python
>>> G = nx.path_graph(4)
>>> G.add_path([10, 11, 12])
>>> sorted(nx.connected_components(G), key = len, reverse=True)
[[0, 1, 2, 3], [10, 11, 12]]
```

**connected_component_subgraphs**

`connected_component_subgraphs(G, copy=True)`  
Generate connected components as subgraphs.

**Parameters**  
G : NetworkX graph  
An undirected graph.

**Returns**  
comp : generator  
A generator of graphs, one for each connected component of G.

`copy` : bool (default=True)  
If True make a copy of the graph attributes

**See Also:**

`connected_components`

Notes

For undirected graphs only. Graph, node, and edge attributes are copied to the subgraphs by default.

Examples

```python
>>> G = nx.path_graph(4)
>>> G.add_edge(5,6)
>>> graphs = list(nx.connected_component_subgraphs(G))
```

**node_connected_component**

`node_connected_component(G, n)`  
Return the nodes in the component 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.

4.11.2 Strong connectivity

Strongly connected components.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>is_strongly_connected(G)</td>
<td>Test directed graph for strong connectivity.</td>
</tr>
<tr>
<td>number_strongly_connected_components(G)</td>
<td>Return number of strongly connected components in graph.</td>
</tr>
<tr>
<td>strongly_connected_components(G)</td>
<td>Generate nodes in strongly connected components of graph.</td>
</tr>
<tr>
<td>strongly_connected_component_subgraphs(G[, copy])</td>
<td>Generate strongly connected components as subgraphs.</td>
</tr>
<tr>
<td>strongly_connected_components_recursive(G)</td>
<td>Generate nodes in strongly connected components of graph.</td>
</tr>
<tr>
<td>kosaraju_strongly_connected_components(G[, ...])</td>
<td>Generate nodes in strongly connected components of graph.</td>
</tr>
<tr>
<td>condensation(G[, scc])</td>
<td>Returns the condensation of G.</td>
</tr>
</tbody>
</table>

**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.

**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.

strongly_connected_components

**strongly_connected_components** (*G*)
Generate nodes in strongly connected components of graph.

Parameters

- **G**: NetworkX Graph
  An directed graph.

Returns

- **comp**: generator of lists
  A list of nodes for each strongly connected component of G.

Raises

- NetworkXNotImplemented: If G is undirected.

See Also:
connected_components, weakly_connected_components

Notes

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

References

[R208], [R209]

strongly_connected_component_subgraphs

**strongly_connected_component_subgraphs** (*G, copy=True*)
Generate strongly connected components as subgraphs.

Parameters

- **G**: NetworkX Graph
  A graph.

Returns

- **comp**: generator of lists
  A list of graphs, one for each strongly connected component of G.

- **copy**: boolean
  If copy is True, Graph, node, and edge attributes are copied to the subgraphs.

See Also:
connected_component_subgraphs
strongly_connected_components_recursive

\[ \text{strongly_connected_components_recursive}(G) \]

Generate nodes in strongly connected components of graph.

Recursive version of algorithm.

**Parameters**  
\( G \) : NetworkX Graph  
An directed graph.

**Returns**  
\( \text{comp} \) : generator of lists  
A list of nodes for each component of \( G \). The list is ordered from largest connected component to smallest.

**Raises**  
\text{NetworkXNotImplemented} : If \( G \) is undirected

**See Also:**  
connected_components

**Notes**

Uses Tarjan’s algorithm with Nuutila’s modifications.

**References**

[R210], [R211]

kosaraju_strongly_connected_components

\[ \text{kosaraju_strongly_connected_components}(G, \text{source}=None) \]

Generate nodes in strongly connected components of graph.

**Parameters**  
\( G \) : NetworkX Graph  
An directed graph.

**Returns**  
\( \text{comp} \) : generator of lists  
A list of nodes for each component of \( G \). The list is ordered from largest connected component to smallest.

** Raises**  
\text{NetworkXNotImplemented} : If \( G \) is undirected

**See Also:**  
connected_components

**Notes**

Uses Kosaraju’s algorithm.
condensation

condensation \( (G, scc=None) \)
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 DiGraph
A directed graph.

scc: list or generator (optional, default=None)
Strongly connected components. If provided, the elements in scc must
partition the nodes in G. If not provided, it will be calculated as
scc=nx.strongly_connected_components(G).

Returns C : NetworkX DiGraph
The condensation graph C of G. The node labels are integers corresponding to the index
of the component in the list of strongly connected components of G. C has a graph
attribute named ‘mapping’ with a dictionary mapping the original nodes to the nodes in
C to which they belong. Each node in C also has a node attribute ‘members’ with the
list of original nodes in G that form the SCC that the node in C represents.

Raises NetworkXNotImplemented: If G is not directed

Notes

After contracting all strongly connected components to a single node, the resulting graph is a directed acyclic
graph.

4.11.3 Weak connectivity

Weakly connected components.

<table>
<thead>
<tr>
<th>is_weakly_connected(G)</th>
<th>Test directed graph for weak connectivity.</th>
</tr>
</thead>
<tbody>
<tr>
<td>number_weakly_connected_components(G)</td>
<td>Return the number of connected components in G.</td>
</tr>
<tr>
<td>weakly_connected_components(G)</td>
<td>Generate weakly connected components of G.</td>
</tr>
<tr>
<td>weakly_connected_component_subgraphs(G[, copy])</td>
<td>Generate weakly connected components as subgraphs.</td>
</tr>
</tbody>
</table>

is_weakly_connected

is_weakly_connected \( (G) \)
Test directed graph for weak connectivity.

A directed graph is weakly connected if, and only if, the graph is connected when the direction of the edge
between nodes is ignored.

Parameters G : NetworkX Graph
A directed graph.

Returns connected : bool
True if the graph is weakly connected, False otherwise.
See Also:

is_strongly_connected, is_semiconnected, is_connected

Notes

For directed graphs only.

number_weakly_connected_components

number_weakly_connected_components(G)

Return the number of connected components in G. For directed graphs only.

weakly_connected_components

weakly_connected_components(G)

Generate weakly connected components of G.

weakly_connected_component_subgraphs

weakly_connected_component_subgraphs(G, copy=True)

Generate weakly connected components as subgraphs.

Parameters

- **G**: NetworkX Graph
  - A directed graph.
- **copy**: bool
  - If copy is True, graph, node, and edge attributes are copied to the subgraphs.

### 4.11.4 Attracting components

Attracting components.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>is_attracting_component(G)</td>
<td>Returns True if G consists of a single attracting component.</td>
</tr>
<tr>
<td>number_attracting_components(G)</td>
<td>Returns the number of attracting components in G.</td>
</tr>
<tr>
<td>attracting_components(G)</td>
<td>Generates a list of attracting components in G.</td>
</tr>
<tr>
<td>attracting_component_subgraphs(G, copy)</td>
<td>Generates a list of attracting component subgraphs from G.</td>
</tr>
</tbody>
</table>

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

**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

**attracting_components**

**attracting_components** \((G)\)

Generates 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 : generator of 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

**attracting_component_subgraphs**

**attracting_component_subgraphs** \((G, copy=True)\)

Generates 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 \(G\).

- copy : bool
  - If copy is True, graph, node, and edge attributes are copied to the subgraphs.
### 4.11.5 Biconnected components

Biconnected components and articulation points.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>is_biconnected(G)</code></td>
<td>Return <code>True</code> if the graph is biconnected, <code>False</code> otherwise.</td>
</tr>
<tr>
<td><code>biconnected_components(G)</code></td>
<td>Return a generator of sets of nodes, one set for each biconnected component.</td>
</tr>
<tr>
<td><code>biconnected_component_edges(G)</code></td>
<td>Return a generator of lists of edges, one list for each biconnected component.</td>
</tr>
<tr>
<td><code>biconnected_component_subgraphs(G[, copy])</code></td>
<td>Return a generator of graphs, one graph for each biconnected component.</td>
</tr>
<tr>
<td><code>articulation_points(G)</code></td>
<td>Return a generator of articulation points, or cut vertices, of a graph.</td>
</tr>
</tbody>
</table>

#### is_biconnected

**is_biconnected** (*G*)

Return `True` if the graph is biconnected, `False` otherwise.

A graph is biconnected if, and only if, it cannot be disconnected by removing only one node (and all edges incident on that node). If removing a node increases the number of disconnected components in the graph, that node is called an articulation point, or cut vertex. A biconnected graph has no articulation points.

**Parameters**

- `G` : NetworkX Graph
  An undirected graph.

**Returns**

- `biconnected` : bool
  True if the graph is biconnected, False otherwise.

**Raises**

- `NetworkXNotImplemented`
  If the input graph is not undirected.

**See Also**

- `biconnected_components, articulation_points, biconnected_component_edges, biconnected_component_subgraphs`

**Notes**

The algorithm to find articulation points and biconnected components is implemented using a non-recursive depth-first-search (DFS) that keeps track of the highest level that back edges reach in the DFS tree. A node \( n \) is an articulation point if, and only if, there exists a subtree rooted at \( n \) such that there is no back edge from any successor of \( n \) that links to a predecessor of \( n \) in the DFS tree. By keeping track of all the edges traversed by the DFS we can obtain the biconnected components because all edges of a bicomponent will be traversed consecutively between articulation points.

**References**

[R207]
Examples

```python
>>> G = nx.path_graph(4)
>>> print(nx.is_biconnected(G))
False
>>> G.add_edge(0, 3)
>>> print(nx.is_biconnected(G))
True
```

*biconnected_components*

*biconnected_components*(G)
Return a generator of sets of nodes, one set for each biconnected component of the graph

Biconnected components are maximal subgraphs such that the removal of a node (and all edges incident on that node) will not disconnect the subgraph. Note that nodes may be part of more than one biconnected component. Those nodes are articulation points, or cut vertices. The removal of articulation points will increase the number of connected components of the graph.

Notice that by convention a dyad is considered a biconnected component.

**Parameters**

- **G** : NetworkX Graph
  An undirected graph.

**Returns**

- **nodes** : generator
  Generator of sets of nodes, one set for each biconnected component.

**Raises**

- **NetworkXNotImplemented** :
  If the input graph is not undirected.

**See Also:**

- `is_biconnected`
- `articulation_points`
- `biconnected_component_edges`
- `biconnected_component_subgraphs`

**Notes**

The algorithm to find articulation points and biconnected components is implemented using a non-recursive depth-first-search (DFS) that keeps track of the highest level that back edges reach in the DFS tree. A node \( n \) is an articulation point if, and only if, there exists a subtree rooted at \( n \) such that there is no back edge from any successor of \( n \) that links to a predecessor of \( n \) in the DFS tree. By keeping track of all the edges traversed by the DFS we can obtain the biconnected components because all edges of a bicomponent will be traversed consecutively between articulation points.

**References**

[R206]

**Examples**
>>> G = nx.barbell_graph(4,2)
>>> print(nx.is_biconnected(G))
False
>>> components = nx.biconnected_components(G)
>>> G.add_edge(2,8)
>>> print(nx.is_biconnected(G))
True
>>> components = nx.biconnected_components(G)

biconnected_component_edges

biconnected_component_edges(G)

Return a generator of lists of edges, one list for each biconnected component of the input graph.

Biconnected components are maximal subgraphs such that the removal of a node (and all edges incident on that node) will not disconnect the subgraph. Note that nodes may be part of more than one biconnected component. Those nodes are articulation points, or cut vertices. However, each edge belongs to one, and only one, biconnected component.

Notice that by convention a dyad is considered a biconnected component.

Parameters  
G : NetworkX Graph

An undirected graph.

Returns  
edges : generator of lists

Generator of lists of edges, one list for each bicomponent.

Raises  
NetworkXNotImplemented :

If the input graph is not undirected.

See Also:

is_biconnected,  biconnected_components,  articulation_points,
                 biconnected_component_subgraphs

Notes

The algorithm to find articulation points and biconnected components is implemented using a non-recursive depth-first-search (DFS) that keeps track of the highest level that back edges reach in the DFS tree. A node \( n \) is an articulation point if, and only if, there exists a subtree rooted at \( n \) such that there is no back edge from any successor of \( n \) that links to a predecessor of \( n \) in the DFS tree. By keeping track of all the edges traversed by the DFS we can obtain the biconnected components because all edges of a bicomponent will be traversed consecutively between articulation points.

References

[R204]

Examples
>>> G = nx.barbell_graph(4,2)
>>> print(nx.is_biconnected(G))
False
>>> components = nx.biconnected_component_edges(G)
>>> G.add_edge(2,8)
>>> print(nx.is_biconnected(G))
True
>>> components = nx.biconnected_component_edges(G)

biconnected_component_subgraphs

biconnected_component_subgraphs(G, copy=True)
Return a generator of graphs, one graph for each biconnected component of the input graph.

Biconnected components are maximal subgraphs such that the removal of a node (and all edges incident on that
node) will not disconnect the subgraph. Note that nodes may be part of more than one biconnected component.
Those nodes are articulation points, or cut vertices. The removal of articulation points will increase the number
of connected components of the graph.

Notice that by convention a dyad is considered a biconnected component.

Parameters  G : NetworkX Graph
   An undirected graph.

Returns  graphs : generator
   Generator of graphs, one graph for each biconnected component.

Raises  NetworkXNotImplemented :
   If the input graph is not undirected.

See Also:
is_biconnected, articulation_points, biconnected_component_edges,
biconnected_components

Notes

The algorithm to find articulation points and biconnected components is implemented using a non-recursive
depth-first-search (DFS) that keeps track of the highest level that back edges reach in the DFS tree. A node n
is an articulation point if, and only if, there exists a subtree rooted at n such that there is no back edge from
any successor of n that links to a predecessor of n in the DFS tree. By keeping track of all the edges traversed
by the DFS we can obtain the biconnected components because all edges of a bicomponent will be traversed
consecutively between articulation points.

Graph, node, and edge attributes are copied to the subgraphs.

References

[R205]
Examples

```python
>>> G = nx.barbell_graph(4, 2)
>>> print(nx.is_biconnected(G))
False
>>> subgraphs = list(nx.biconnected_component_subgraphs(G))
```

articulation_points

**articulation_points(G)**

Return a generator of articulation points, or cut vertices, of a graph.

An articulation point or cut vertex is any node whose removal (along with all its incident edges) increases the number of connected components of a graph. An undirected connected graph without articulation points is biconnected. Articulation points belong to more than one biconnected component of a graph.

Notice that by convention a dyad is considered a biconnected component.

**Parameters**

- **G** : NetworkX Graph
  
  An undirected graph.

**Returns**

- **articulation points** : generator
  
  generator of nodes

**Raises**

- **NetworkXNotImplemented** : If the input graph is not undirected.

**See Also**

- `is_biconnected`
- `biconnected_components`
- `biconnected_component_edges`
- `biconnected_component_subgraphs`

**Notes**

The algorithm to find articulation points and biconnected components is implemented using a non-recursive depth-first-search (DFS) that keeps track of the highest level that back edges reach in the DFS tree. A node $n$ is an articulation point if, and only if, there exists a subtree rooted at $n$ such that there is no back edge from any successor of $n$ that links to a predecessor of $n$ in the DFS tree. By keeping track of all the edges traversed by the DFS we can obtain the biconnected components because all edges of a bicomponent will be traversed consecutively between articulation points.

**References**

[R203]

**Examples**

```python
>>> G = nx.barbell_graph(4, 2)
>>> print(nx.is_biconnected(G))
False
>>> list(nx.articulation_points(G))
[6, 5, 4, 3]
```
>>> G.add_edge(2,8)
>>> print(nx.is_biconnected(G))
True
>>> list(nx.articulation_points(G))
[]

4.11.6 Semiconnectedness

Semiconnectedness.

\[
\text{is\_semiconnected}(G) \quad \text{Return True if the graph is semiconnected, False otherwise.}
\]

is_semiconnected

\text{is\_semiconnected}(G) \quad \text{Return True if the graph is semiconnected, False otherwise.}

A graph is semiconnected if, and only if, for any pair of nodes, either one is reachable from the other, or they are mutually reachable.

\text{Parameters}\quad \text{G : NetworkX graph}

A directed graph.

\text{Returns}\quad \text{semiconnected : bool}

True if the graph is semiconnected, False otherwise.

\text{Raises}\quad \text{NetworkXNotImplemented :}

If the input graph is not directed.

\text{NetworkXPointlessConcept :}

If the graph is empty.

\text{See Also:}

\text{is\_strongly\_connected, is\_weakly\_connected}

\text{Examples}

>>> G=nx.path_graph(4,create_using=nx.DiGraph())
>>> print(nx.is_semiconnected(G))
True
>>> G=nx.DiGraph([(1, 2), (3, 2)])
>>> print(nx.is_semiconnected(G))
False

4.12 Connectivity

Connectivity and cut algorithms
### 4.12.1 Flow-based Connectivity

Flow based connectivity algorithms

- `average_node_connectivity(G[, flow_func])` Returns the average connectivity of a graph G.
- `all_pairs_node_connectivity(G[, nbunch, ...])` Compute node connectivity between all pairs of nodes of G.
- `edge_connectivity(G[, s, t, flow_func])` Returns the edge connectivity of the graph or digraph G.
- `local_edge_connectivity(G, u, v[, ...])` Returns local edge connectivity for nodes s and t in G.
- `local_node_connectivity(G, s, t[, ...])` Computes local node connectivity for nodes s and t.
- `node_connectivity(G[, s, t, flow_func])` Returns node connectivity for a graph or digraph G.

#### average_node_connectivity

**average_node_connectivity**(G, flow_func=None)

Returns the average connectivity of a graph G.

The average connectivity $\bar{\kappa}$ of a graph G is the average of local node connectivity over all pairs of nodes of G [R212].

$$\bar{\kappa}(G) = \frac{\sum_{u,v} \kappa_G(u, v)}{\binom{n}{2}}$$

**Parameters**

- `G`: NetworkX graph
  Undirected graph
- `flow_func`: function
  A function for computing the maximum flow among a pair of nodes. The function has to accept at least three parameters: a Digraph, a source node, and a target node. And return a residual network that follows NetworkX conventions (see `maximum_flow()` for details). If `flow_func` is None, the default maximum flow function (`edmonds_karp()`) is used. See `local_node_connectivity()` for details. The choice of the default function may change from version to version and should not be relied on. Default value: None.

**Returns**

- `K`: float
  Average node connectivity

**See Also:**

- `local_node_connectivity()`, `node_connectivity()`, `edge_connectivity()`, `maximum_flow()`, `edmonds_karp()`, `preflow_push()`, `shortest_augmenting_path()`

**References**

[R212]

#### all_pairs_node_connectivity

**all_pairs_node_connectivity**(G, nbunch=None, flow_func=None)

Compute node connectivity between all pairs of nodes of G.

**Parameters**

- `G`: NetworkX graph
Undirected graph

**nbunch**: container

Container of nodes. If provided node connectivity will be computed only over pairs of nodes in nbunch.

**flow_func**: function

A function for computing the maximum flow among a pair of nodes. The function has to accept at least three parameters: a Digraph, a source node, and a target node. And return a residual network that follows NetworkX conventions (see `maximum_flow()` for details). If flow_func is None, the default maximum flow function (`edmonds_karp()`) is used. See below for details. The choice of the default function may change from version to version and should not be relied on. Default value: None.

**Returns**  
`all_pairs`: dict  
A dictionary with node connectivity between all pairs of nodes in G, or in nbunch if provided.

**See Also:**

local_node_connectivity(), edge_connectivity(), local_edge_connectivity(), maximum_flow(), edmonds_karp(), preflow_push(), shortest_augmenting_path()

---

**edge_connectivity**

**edge_connectivity**(G, s=None, t=None, flow_func=None)  

Returns the edge connectivity of the graph or digraph G.

The edge connectivity is equal to the minimum number of edges that must be removed to disconnect G or render it trivial. If source and target nodes are provided, this function returns the local edge connectivity: the minimum number of edges that must be removed to break all paths from source to target in G.

**Parameters**

- **G**: NetworkX graph
  - Undirected or directed graph
- **s**: node
  - Source node. Optional. Default value: None.
- **t**: node
  - Target node. Optional. Default value: None.
- **flow_func**: function
  - A function for computing the maximum flow among a pair of nodes. The function has to accept at least three parameters: a Digraph, a source node, and a target node. And return a residual network that follows NetworkX conventions (see `maximum_flow()` for details). If flow_func is None, the default maximum flow function (`edmonds_karp()`) is used. See below for details. The choice of the default function may change from version to version and should not be relied on. Default value: None.

**Returns**

- **K**: integer
  - Edge connectivity for G, or local edge connectivity if source and target were provided

**See Also:**

local_edge_connectivity(), local_node_connectivity(), node_connectivity(), maximum_flow(), edmonds_karp(), preflow_push(), shortest_augmenting_path()
Notes

This is a flow based implementation of global edge connectivity. For undirected graphs the algorithm works by finding a ‘small’ dominating set of nodes of G (see algorithm 7 in [R213]) and computing local maximum flow (see \texttt{local_edge_connectivity()}) between an arbitrary node in the dominating set and the rest of nodes in it. This is an implementation of algorithm 6 in [R213]. For directed graphs, the algorithm does \( n \) calls to the maximum flow function. This is an implementation of algorithm 8 in [R213].

References

[R213]

Examples

```python
>>> # Platonic icosahedral graph is 5-edge-connected
>>> G = nx.icosahedral_graph()
>>> nx.edge_connectivity(G)
5
```

You can use alternative flow algorithms for the underlying maximum flow computation. In dense networks the algorithm \texttt{shortest_augmenting_path()} will usually perform better than the default \texttt{edmonds_karp()}, which is faster for sparse networks with highly skewed degree distributions. Alternative flow functions have to be explicitly imported from the flow package.

```python
>>> from networkx.algorithms.flow import shortest_augmenting_path
>>> nx.edge_connectivity(G, flow_func=shortest_augmenting_path)
5
```

If you specify a pair of nodes (source and target) as parameters, this function returns the value of local edge connectivity.

```python
>>> nx.edge_connectivity(G, 3, 7)
5
```

If you need to perform several local computations among different pairs of nodes on the same graph, it is recommended that you reuse the data structures used in the maximum flow computations. See \texttt{local_edge_connectivity()} for details.

\texttt{local_edge_connectivity}

\texttt{local_edge_connectivity}(G, u, v, flow_func=None, auxiliary=None, residual=None, cutoff=None)

Returns local edge connectivity for nodes \( s \) and \( t \) in \( G \).

Local edge connectivity for two nodes \( s \) and \( t \) is the minimum number of edges that must be removed to disconnect them.

This is a flow based implementation of edge connectivity. We compute the maximum flow on an auxiliary digraph build from the original network (see below for details). This is equal to the local edge connectivity because the value of a maximum \( s \)-\( t \)-flow is equal to the capacity of a minimum \( s \)-\( t \)-cut (Ford and Fulkerson theorem) [R214].

Parameters

\( G \) : NetworkX graph

\( s \) : node
Source node t : node
Target node

flow_func : function
A function for computing the maximum flow among a pair of nodes. The function has to accept at least three parameters: a Digraph, a source node, and a target node. And return a residual network that follows NetworkX conventions (see maximum_flow() for details). If flow_func is None, the default maximum flow function (edmonds_karp()) is used. See below for details. The choice of the default function may change from version to version and should not be relied on. Default value: None.

auxiliary : NetworkX DiGraph
Auxiliary digraph for computing flow based edge connectivity. If provided it will be reused instead of recreated. Default value: None.

residual : NetworkX DiGraph
Residual network to compute maximum flow. If provided it will be reused instead of recreated. Default value: None.

cutoff : integer, float
If specified, the maximum flow algorithm will terminate when the flow value reaches or exceeds the cutoff. This is only for the algorithms that support the cutoff parameter: edmonds_karp() and shortest_augmenting_path(). Other algorithms will ignore this parameter. Default value: None.

Returns K : integer
local edge connectivity for nodes s and t.

See Also:
edge_connectivity(), local_node_connectivity(), node_connectivity(), maximum_flow(), edmonds_karp(), preflow_push(), shortest_augmenting_path()

Notes
This is a flow based implementation of edge connectivity. We compute the maximum flow using, by default, the edmonds_karp() algorithm on an auxiliary digraph build from the original input graph:

If the input graph is undirected, we replace each edge (u,’v’) with two reciprocal arcs (u, v) and (v, u) and then we set the attribute ‘capacity’ for each arc to 1. If the input graph is directed we simply add the ‘capacity’ attribute. This is an implementation of algorithm 1 in [R214].

The maximum flow in the auxiliary network is equal to the local edge connectivity because the value of a maximum s-t-flow is equal to the capacity of a minimum s-t-cut (Ford and Fulkerson theorem).

References

[R214]
Examples

This function is not imported in the base NetworkX namespace, so you have to explicitly import it from the
connectivity package:

```python
>>> from networkx.algorithms.connectivity import local_edge_connectivity
```

We use in this example the platonic icosahedral graph, which has edge connectivity 5.

```python
>>> G = nx.icosahedral_graph()
>>> local_edge_connectivity(G, 0, 6)
5
```

If you need to compute local connectivity on several pairs of nodes in the same graph, it is recommended that
you reuse the data structures that NetworkX uses in the computation: the auxiliary digraph for edge connectivity,
and the residual network for the underlying maximum flow computation.

Example of how to compute local edge connectivity among all pairs of nodes of the platonic icosahedral graph
reusing the data structures.

```python
>>> import itertools

>>> # You also have to explicitly import the function for
>>> # building the auxiliary digraph from the connectivity package
>>> from networkx.algorithms.connectivity import build_auxiliary_edge_connectivity

>>> H = build_auxiliary_edge_connectivity(G)

>>> # And the function for building the residual network from the
>>> # flow package
>>> from networkx.algorithms.flow import build_residual_network

>>> R = build_residual_network(H, 'capacity')

>>> result = dict.fromkeys(G, dict())

>>> # Reuse the auxiliary digraph and the residual network by passing them
>>> # as parameters
>>> for u, v in itertools.combinations(G, 2):
...     k = local_edge_connectivity(G, u, v, auxiliary=H, residual=R)
...     result[u][v] = k
>>> all(result[u][v] == 5 for u, v in itertools.combinations(G, 2))
True
```

You can also use alternative flow algorithms for computing edge connectivity. For instance, in dense
networks the algorithm `shortest_augmenting_path()` will usually perform better than the default
edmonds_karp() which is faster for sparse networks with highly skewed degree distributions. Alternative
flow functions have to be explicitly imported from the flow package.

```python
>>> from networkx.algorithms.flow import shortest_augmenting_path

>>> local_edge_connectivity(G, 0, 6, flow_func=shortest_augmenting_path)
5
```

**local_node_connectivity**

`local_node_connectivity(G, s, t, flow_func=None, auxiliary=None, residual=None, cutoff=None)`

Computes local node connectivity for nodes `s` and `t`.

Local node connectivity for two non adjacent nodes `s` and `t` is the minimum number of nodes that must be
removed (along with their incident edges) to disconnect them.

This is a flow based implementation of node connectivity. We compute the maximum flow on an auxiliary
digraph build from the original input graph (see below for details).
**Parameters**

- **G**: NetworkX graph
  - Undirected graph
- **s**: node
  - Source node
- **t**: node
  - Target node
- **flow_func**: function
  - A function for computing the maximum flow among a pair of nodes. The function has to accept at least three parameters: a Digraph, a source node, and a target node. And return a residual network that follows NetworkX conventions (see `maximum_flow()` for details). If `flow_func` is None, the default maximum flow function (`edmonds_karp()`) is used. See below for details. The choice of the default function may change from version to version and should not be relied on. Default value: None.

- **auxiliary**: NetworkX DiGraph
  - Auxiliary digraph to compute flow based node connectivity. It has to have a graph attribute called mapping with a dictionary mapping node names in G and in the auxiliary digraph. If provided it will be reused instead of recreated. Default value: None.

- **residual**: NetworkX DiGraph
  - Residual network to compute maximum flow. If provided it will be reused instead of recreated. Default value: None.

- **cutoff**: integer, float
  - If specified, the maximum flow algorithm will terminate when the flow value reaches or exceeds the cutoff. This is only for the algorithms that support the cutoff parameter: `edmonds_karp()` and `shortest_augmenting_path()`. Other algorithms will ignore this parameter. Default value: None.

**Returns**

- **K**: integer
  - local node connectivity for nodes s and t

**See Also**

- `local_edge_connectivity()`, `node_connectivity()`, `minimum_node_cut()`, `maximum_flow()`, `edmonds_karp()`, `preflow_push()`, `shortest_augmenting_path()`

**Notes**

This is a flow based implementation of node connectivity. We compute the maximum flow using, by default, the `edmonds_karp()` algorithm (see: `maximum_flow()`) on an auxiliary digraph build from the original input graph:

For an undirected graph G having $n$ nodes and $m$ edges we derive a directed graph H with $2n$ nodes and $2m + n$ arcs by replacing each original node $v$ with two nodes $v_A$, $v_B$ linked by an (internal) arc in H. Then for each edge $(u, v)$ in G we add two arcs $(u_B, v_A)$ and $(v_B, u_A)$ in H. Finally we set the attribute capacity = 1 for each arc in H [R216].

For a directed graph G having $n$ nodes and $m$ arcs we derive a directed graph H with $2n$ nodes and $m + n$ arcs by replacing each original node $v$ with two nodes $v_A$, $v_B$ linked by an (internal) arc $(v_A, v_B)$ in H. Then for each arc $(u, v)$ in G we add one arc $(u_B, v_A)$ in H. Finally we set the attribute capacity = 1 for each arc in H.
This is equal to the local node connectivity because the value of a maximum s-t-flow is equal to the capacity of a minimum s-t-cut.

References

[R216]

Examples

This function is not imported in the base NetworkX namespace, so you have to explicitly import it from the connectivity package:

```python
>>> from networkx.algorithms.connectivity import local_node_connectivity
```

We use in this example the platonic icosahedral graph, which has node connectivity 5.

```python
>>> G = nx.icosahedral_graph()
>>> local_node_connectivity(G, 0, 6)
5
```

If you need to compute local connectivity on several pairs of nodes in the same graph, it is recommended that you reuse the data structures that NetworkX uses in the computation: the auxiliary digraph for node connectivity, and the residual network for the underlying maximum flow computation.

Example of how to compute local node connectivity among all pairs of nodes of the platonic icosahedral graph reusing the data structures.

```python
>>> import itertools

>>> # You also have to explicitly import the function for
>>> # building the auxiliary digraph from the connectivity package
>>> from networkx.algorithms.connectivity import (...
>>> # flow package
>>> from networkx.algorithms.flow import build_residual_network

>>> H = build_auxiliary_node_connectivity(G)

>>> # And the function for building the residual network from the
>>> # flow package
>>> R = build_residual_network(H, 'capacity')

>>> result = dict.fromkeys(G, dict())

>>> # Reuse the auxiliary digraph and the residual network by passing them
>>> # as parameters
>>> for u, v in itertools.combinations(G, 2):
...     k = local_node_connectivity(G, u, v, auxiliary=H, residual=R)
...     result[u][v] = k
>>> all(result[u][v] == 5 for u, v in itertools.combinations(G, 2))
True
```

You can also use alternative flow algorithms for computing node connectivity. For instance, in dense networks the algorithm `shortest_augmenting_path()` will usually perform better than the default `edmonds_karp()` which is faster for sparse networks with highly skewed degree distributions. Alternative flow functions have to be explicitly imported from the flow package.

```python
>>> from networkx.algorithms.flow import shortest_augmenting_path

>>> local_node_connectivity(G, 0, 6, flow_func=shortest_augmenting_path)
5
```
node_connectivity

node_connectivity \((G, s=None, t=None, flow_func=None)\)

Returns node connectivity for a graph or digraph \(G\).

Node connectivity is equal to the minimum number of nodes that must be removed to disconnect \(G\) or render it trivial. If source and target nodes are provided, this function returns the local node connectivity: the minimum number of nodes that must be removed to break all paths from source to target in \(G\).

**Parameters**

- \(G\) : NetworkX graph
  - Undirected graph
- \(s\) : node
  - Source node. Optional. Default value: None.
- \(t\) : node
  - Target node. Optional. Default value: None.
- \(flow_func\) : function
  - A function for computing the maximum flow among a pair of nodes. The function has to accept at least three parameters: a Digraph, a source node, and a target node. And return a residual network that follows NetworkX conventions (see maximum_flow() for details). If \(flow_func\) is None, the default maximum flow function (edmonds_karp()) is used. See below for details. The choice of the default function may change from version to version and should not be relied on. Default value: None.

**Returns**

- \(K\) : integer
  - Node connectivity of \(G\), or local node connectivity if source and target are provided.

**See Also:**

- local_node_connectivity(), edge_connectivity(), maximum_flow(), edmonds_karp(), preflow_push(), shortest_augmenting_path()

**Notes**

This is a flow based implementation of node connectivity. The algorithm works by solving \(O((n-\delta-1+\delta(\delta-1)/2)\) maximum flow problems on an auxiliary digraph. Where \(\delta\) is the minimum degree of \(G\). For details about the auxiliary digraph and the computation of local node connectivity see local_node_connectivity(). This implementation is based on algorithm 11 in [R217].

**References**

[R217]

**Examples**

```python
>>> # Platonic icosahedral graph is 5-node-connected
>>> G = nx.icosahedral_graph()
>>> nx.node_connectivity(G)
5
```
You can use alternative flow algorithms for the underlying maximum flow computation. In dense networks the algorithm `shortest_augmenting_path()` will usually perform better than the default `edmonds_karp()`, which is faster for sparse networks with highly skewed degree distributions. Alternative flow functions have to be explicitly imported from the flow package.

```python
>>> from networkx.algorithms.flow import shortest_augmenting_path
>>> nx.node_connectivity(G, flow_func=shortest_augmenting_path)
5
```

If you specify a pair of nodes (source and target) as parameters, this function returns the value of local node connectivity.

```python
>>> nx.node_connectivity(G, 3, 7)
5
```

If you need to perform several local computations among different pairs of nodes on the same graph, it is recommended that you reuse the data structures used in the maximum flow computations. See `local_node_connectivity()` for details.

### 4.12.2 Flow-based Minimum Cuts

Flow based cut algorithms

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>minimum_edge_cut(G[, s, t, flow_func])</code></td>
<td>Returns a set of edges of minimum cardinality that disconnects G.</td>
</tr>
<tr>
<td><code>minimum_node_cut(G[, s, t, flow_func])</code></td>
<td>Returns a set of nodes of minimum cardinality that disconnects G.</td>
</tr>
<tr>
<td><code>minimum_st_edge_cut(G, s, t[, flow_func, ...])</code></td>
<td>Returns the edges of the cut-set of a minimum (s, t)-cut.</td>
</tr>
<tr>
<td><code>minimum_st_node_cut(G, s, t[, flow_func, ...])</code></td>
<td>Returns a set of nodes of minimum cardinality that disconnect source</td>
</tr>
</tbody>
</table>

#### minimum_edge_cut

`minimum_edge_cut(G, s=None, t=None, flow_func=None)`

Returns a set of edges of minimum cardinality that disconnects G.

If source and target nodes are provided, this function returns the set of edges of minimum cardinality that, if removed, would break all paths among source and target in G. If not, it returns a set of edges of minimum cardinality that disconnects G.

**Parameters**

- `G` : NetworkX graph
- `s` : node
  - Source node. Optional. Default value: None.
- `t` : node
  - Target node. Optional. Default value: None.
- `flow_func` : function
  - A function for computing the maximum flow among a pair of nodes. The function has to accept at least three parameters: a Digraph, a source node, and a target node. And return a residual network that follows NetworkX conventions (see `maximum_flow()` for details). If `flow_func` is None, the default maximum flow function (`edmonds_karp()`) is used. See below for details. The choice of the default function may change from version to version and should not be relied on. Default value: None.

**Returns**

- `cutset` : set

---

4.12. Connectivity
Set of edges that, if removed, would disconnect G. If source and target nodes are provided, the set contains the edges that if removed, would destroy all paths between source and target.

See Also:
minimum_st_edge_cut(), minimum_node_cut(), stoer_wagner(), node_connectivity(), edge_connectivity(), maximum_flow(), edmonds_karp(), preflow_push(), shortest_augmenting_path()

Notes
This is a flow based implementation of minimum edge cut. For undirected graphs the algorithm works by finding a ‘small’ dominating set of nodes of G (see algorithm 7 in [R218]) and computing the maximum flow between an arbitrary node in the dominating set and the rest of nodes in it. This is an implementation of algorithm 6 in [R218]. For directed graphs, the algorithm does n calls to the max flow function. It is an implementation of algorithm 8 in [R218].

References
[R218]

Examples

>>> # Platonic icosahedral graph has edge connectivity 5
>>> G = nx.icosahedral_graph()
>>> len(nx.minimum_edge_cut(G))
5

You can use alternative flow algorithms for the underlying maximum flow computation. In dense networks the algorithm shortest_augmenting_path() will usually perform better than the default edmonds_karp(), which is faster for sparse networks with highly skewed degree distributions. Alternative flow functions have to be explicitly imported from the flow package.

>>> from networkx.algorithms.flow import shortest_augmenting_path
>>> len(nx.minimum_edge_cut(G, flow_func=shortest_augmenting_path))
5

If you specify a pair of nodes (source and target) as parameters, this function returns the value of local edge connectivity.

>>> nx.edge_connectivity(G, 3, 7)
5

If you need to perform several local computations among different pairs of nodes on the same graph, it is recommended that you reuse the data structures used in the maximum flow computations. See local_edge_connectivity() for details.

minimum_node_cut

minimum_node_cut(G, s=None, t=None, flow_func=None)

Returns a set of nodes of minimum cardinality that disconnects G.
If source and target nodes are provided, this function returns the set of nodes of minimum cardinality that, if removed, would destroy all paths among source and target in G. If not, it returns a set of nodes of minimum cardinality that disconnects G.

**Parameters**

- **G**: NetworkX graph
- **s**: node
  
  Source node. Optional. Default value: None.
- **t**: node
  
  Target node. Optional. Default value: None.
- **flow_func**: function
  
  A function for computing the maximum flow among a pair of nodes. The function has to accept at least three parameters: a Digraph, a source node, and a target node. And return a residual network that follows NetworkX conventions (see `maximum_flow()` for details). If `flow_func` is None, the default maximum flow function (`edmonds_karp()`) is used. See below for details. The choice of the default function may change from version to version and should not be relied on. Default value: None.

**Returns**

- **cutset**: set
  
  Set of nodes that, if removed, would disconnect G. If source and target nodes are provided, the set contains the nodes that if removed, would destroy all paths between source and target.

**See Also:**

- `minimum_st_node_cut()`, `minimum_cut()`, `minimum_edge_cut()`, `stoer_wagner()`, `node_connectivity()`, `edge_connectivity()`, `maximum_flow()`, `edmonds_karp()`, `preflow_push()`, `shortest_augmenting_path()`

**Notes**

This is a flow based implementation of minimum node cut. The algorithm is based in solving a number of maximum flow computations to determine the capacity of the minimum cut on an auxiliary directed network that corresponds to the minimum node cut of G. It handles both directed and undirected graphs. This implementation is based on algorithm 11 in [R219].

**References**

[R219]

**Examples**

```python
>>> # Platonic icosahedral graph has node connectivity 5
>>> G = nx.icosahedral_graph()
>>> node_cut = nx.minimum_node_cut(G)
>>> len(node_cut)
5
```

You can use alternative flow algorithms for the underlying maximum flow computation. In dense networks the algorithm `shortest_augmenting_path()` will usually perform better than the default
edmonds_karp(), which is faster for sparse networks with highly skewed degree distributions. Alternative flow functions have to be explicitly imported from the flow package.

```python
>>> from networkx.algorithms.flow import shortest_augmenting_path
>>> node_cut == nx.minimum_node_cut(G, flow_func=shortest_augmenting_path)
True
```

If you specify a pair of nodes (source and target) as parameters, this function returns a local st node cut.

```python
>>> len(nx.minimum_node_cut(G, 3, 7))
5
```

If you need to perform several local st cuts among different pairs of nodes on the same graph, it is recommended that you reuse the data structures used in the maximum flow computations. See `minimum_st_node_cut()` for details.

**minimum_st_edge_cut**

minimum_st_edge_cut(G, s, t, flow_func=None, auxiliary=None, residual=None)

Returns the edges of the cut-set of a minimum (s, t)-cut.

This function returns the set of edges of minimum cardinality that, if removed, would destroy all paths among source and target in G. Edge weights are not considered

**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.
- **auxiliary**: NetworkX DiGraph
  - Auxiliary digraph to compute flow based node connectivity. It has to have a graph attribute called mapping with a dictionary mapping node names in G and in the auxiliary digraph. If provided it will be reused instead of recreated. Default value: None.
- **flow_func**: function
  - A function for computing the maximum flow among a pair of nodes. The function has to accept at least three parameters: a Digraph, a source node, and a target node. And return a residual network that follows NetworkX conventions (see `maximum_flow()` for details). If flow_func is None, the default maximum flow function (edmonds_karp()) is used. See `node_connectivity()` for details. The choice of the default function may change from version to version and should not be relied on. Default value: None.
- **residual**: NetworkX DiGraph
  - Residual network to compute maximum flow. If provided it will be reused instead of recreated. Default value: None.

**Returns**

- **cutset**: set
  - Set of edges that, if removed from the graph, will disconnect it.
See Also:
minimum_cut(), minimum_node_cut(), minimum_edge_cut(), stoer_wagner(),
node_connectivity(), edge_connectivity(), maximum_flow(), edmonds_karp(),
preflow_push(), shortest_augmenting_path()

Examples

This function is not imported in the base NetworkX namespace, so you have to explicitly import it from the
connectivity package:

```python
>>> from networkx.algorithms.connectivity import minimum_st_edge_cut
```

We use in this example the platonic icosahedral graph, which has edge connectivity 5.

```python
>>> G = nx.icosahedral_graph()
>>> len(minimum_st_edge_cut(G, 0, 6))
5
```

If you need to compute local edge cuts on several pairs of nodes in the same graph, it is recommended that you
reuse the data structures that NetworkX uses in the computation: the auxiliary digraph for edge connectivity,
and the residual network for the underlying maximum flow computation.

Example of how to compute local edge cuts among all pairs of nodes of the platonic icosahedral graph reusing
the data structures.

```python
>>> import itertools

>>> # You also have to explicitly import the function for
>>> # building the auxiliary digraph from the connectivity package
>>> from networkx.algorithms.connectivity import (...
>>>    build_auxiliary_edge_connectivity)
>>> H = build_auxiliary_edge_connectivity(G)

>>> # And the function for building the residual network from the
>>> # flow package
>>> from networkx.algorithms.flow import build_residual_network

>>> # Note that the auxiliary digraph has an edge attribute named capacity
>>> R = build_residual_network(H, 'capacity')

>>> result = dict.fromkeys(G, dict())

>>> # Reuse the auxiliary digraph and the residual network by passing them
>>> # as parameters
>>> for u, v in itertools.combinations(G, 2):
>>>    k = len(minimum_st_edge_cut(G, u, v, auxiliary=H, residual=R))
>>>    result[u][v] = k
>>> all(result[u][v] == 5 for u, v in itertools.combinations(G, 2))
True
```

You can also use alternative flow algorithms for computing edge cuts. For instance, in dense networks the algo-
rithm shortest_augmenting_path() will usually perform better than the default edmonds_karp() which is faster for sparse networks with highly skewed degree distributions. Alternative flow functions have to
be explicitly imported from the flow package.

```python
>>> from networkx.algorithms.flow import shortest_augmenting_path

>>> len(minimum_st_edge_cut(G, 0, 6, flow_func=shortest_augmenting_path))
5
```
**minimum_st_node_cut**

**minimum_st_node_cut** *(G, s, t, flow_func=None, auxiliary=None, residual=None)*

Returns a set of nodes of minimum cardinality that disconnect source from target in G.

This function returns the set of nodes of minimum cardinality that, if removed, would destroy all paths among source and target in G.

**Parameters**

- **G**: NetworkX graph
  - **s**: node  
    Source node.
  - **t**: node  
    Target node.
  - **flow_func**: function  
    A function for computing the maximum flow among a pair of nodes. The function has to accept at least three parameters: a Digraph, a source node, and a target node. And return a residual network that follows NetworkX conventions (see `maximum_flow()` for details). If `flow_func` is None, the default maximum flow function (edmonds_karp()) is used. See below for details. The choice of the default function may change from version to version and should not be relied on. Default value: None.
  - **auxiliary**: NetworkX DiGraph  
    Auxiliary digraph to compute flow based node connectivity. It has to have a graph attribute called mapping with a dictionary mapping node names in G and in the auxiliary digraph. If provided it will be reused instead of recreated. Default value: None.
  - **residual**: NetworkX DiGraph  
    Residual network to compute maximum flow. If provided it will be reused instead of recreated. Default value: None.

**Returns**

- **cutset**: set  
  Set of nodes that, if removed, would destroy all paths between source and target in G.

**See Also:**

- `minimum_node_cut()`, `minimum_edge_cut()`, `stoer_wagner()`, `node_connectivity()`, `edge_connectivity()`, `maximum_flow()`, `edmonds_karp()`, `preflow_push()`, `shortest_augmenting_path()`

**Notes**

This is a flow based implementation of minimum node cut. The algorithm is based in solving a number of maximum flow computations to determine the capacity of the minimum cut on an auxiliary directed network that corresponds to the minimum node cut of G. It handles both directed and undirected graphs. This implementation is based on algorithm 11 in [R221].

**References**

[R221]
Examples

This function is not imported in the base NetworkX namespace, so you have to explicitly import it from the connectivity package:

```python
>>> from networkx.algorithms.connectivity import minimum_st_node_cut
```

We use in this example the platonic icosahedral graph, which has node connectivity 5.

```python
>>> G = nx.icosahedral_graph()
>>> len(minimum_st_node_cut(G, 0, 6))
5
```

If you need to compute local st cuts between several pairs of nodes in the same graph, it is recommended that you reuse the data structures that NetworkX uses in the computation: the auxiliary digraph for node connectivity and node cuts, and the residual network for the underlying maximum flow computation.

Example of how to compute local st node cuts reusing the data structures:

```python
>>> # You also have to explicitly import the function for
>>> # building the auxiliary digraph from the connectivity package
>>> from networkx.algorithms.connectivity import (...
...    build_auxiliary_node_connectivity)
>>> H = build_auxiliary_node_connectivity(G)
>>> # And the function for building the residual network from the
>>> # flow package
>>> from networkx.algorithms.flow import build_residual_network
>>> # Note that the auxiliary digraph has an edge attribute named capacity
>>> R = build_residual_network(H, 'capacity')
>>> # Reuse the auxiliary digraph and the residual network by passing them
>>> # as parameters
>>> len(minimum_st_node_cut(G, 0, 6, auxiliary=H, residual=R))
5
```

You can also use alternative flow algorithms for computing minimum st node cuts. For instance, in dense networks the algorithm `shortest_augmenting_path()` will usually perform better than the default `edmonds_karp()` which is faster for sparse networks with highly skewed degree distributions. Alternative flow functions have to be explicitly imported from the flow package.

```python
>>> from networkx.algorithms.flow import shortest_augmenting_path
>>> len(minimum_st_node_cut(G, 0, 6, flow_func=shortest_augmenting_path))
5
```

4.12.3 Stoer-Wagner minimum cut

Stoer-Wagner minimum cut algorithm.

```
stoer_wagner(G[, weight, heap]) Returns the weighted minimum edge cut using the Stoer-Wagner algorithm.
```

```
stoer_wagner (G, weight='weight', heap=<class 'networkx.utils.heaps.BinaryHeap'>) Returns the weighted minimum edge cut using the Stoer-Wagner algorithm.
```

Determine the minimum edge cut of a connected graph using the Stoer-Wagner algorithm. In weighted cases, all weights must be nonnegative.

4.12. Connectivity
The running time of the algorithm depends on the type of heaps used:

<table>
<thead>
<tr>
<th>Type of heap</th>
<th>Running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary heap</td>
<td>$O(n(m + n) \log n)$</td>
</tr>
<tr>
<td>Fibonacci heap</td>
<td>$O(nm + n^2 \log n)$</td>
</tr>
<tr>
<td>Pairing heap</td>
<td>$O(2^{2\sqrt{\log n}} nm + n^2 \log n)$</td>
</tr>
</tbody>
</table>

**Parameters**

- **G**: NetworkX graph
  
  Edges of the graph are expected to have an attribute named by the weight parameter below. If this attribute is not present, the edge is considered to have unit weight.

  - **weight**: string
    
    Name of the weight attribute of the edges. If the attribute is not present, unit weight is assumed. Default value: ‘weight’.

  - **heap**: class
    
    Type of heap to be used in the algorithm. It should be a subclass of MinHeap or implement a compatible interface.

    If a stock heap implementation is to be used, BinaryHeap is recommended over PairingHeap for Python implementations without optimized attribute accesses (e.g., CPython) despite a slower asymptotic running time. For Python implementations with optimized attribute accesses (e.g., PyPy), PairingHeap provides better performance. Default value: BinaryHeap.

**Returns**

- **cut_value**: integer or float
  
  The sum of weights of edges in a minimum cut.

- **partition**: pair of node lists
  
  A partitioning of the nodes that defines a minimum cut.

**Raises**

- **NetworkXNotImplemented**
  
  If the graph is directed or a multigraph.

- **NetworkXError**
  
  If the graph has less than two nodes, is not connected or has a negative-weighted edge.

**Examples**

```python
>>> G = nx.Graph()
>>> G.add_edge('x','a', weight=3)
>>> G.add_edge('x','b', weight=1)
>>> G.add_edge('a','c', weight=3)
>>> G.add_edge('b','c', weight=5)
>>> G.add_edge('b','d', weight=4)
>>> G.add_edge('d','e', weight=2)
>>> G.add_edge('c','y', weight=2)
>>> G.add_edge('e','y', weight=3)
>>> cut_value, partition = nx.stoer_wagner(G)
>>> cut_value
4
```
4.12.4 Utils for flow-based connectivity

Utilities for connectivity package

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>build auxiliary_edge_connectivity(G)</td>
<td>Auxiliary digraph for computing flow based edge connectivity</td>
</tr>
<tr>
<td>build auxiliary_node_connectivity(G)</td>
<td>Creates a directed graph D from an undirected graph G to compute flow connectivity</td>
</tr>
</tbody>
</table>

**build auxiliary edge connectivity**

**build auxiliary edge connectivity** \((G)\)

Auxiliary digraph for computing flow based edge connectivity

If the input graph is undirected, we replace each edge \((u, v)\) with two reciprocal arcs \((u, v)\) and \((v, u)\) and then we set the attribute ‘capacity’ for each arc to 1. If the input graph is directed, we simply add the ‘capacity’ attribute. Part of algorithm 1 in [R223].

**References**

[R223]

**build auxiliary node connectivity**

**build auxiliary node connectivity** \((G)\)

Creates a directed graph D from an undirected graph G to compute flow based node connectivity.

For an undirected graph G having \(n\) nodes and \(m\) edges we derive a directed graph D with \(2n\) nodes and \(2m + n\) arcs by replacing each original node \(v\) with two nodes \(vA, vB\) linked by an (internal) arc in D. Then for each edge \((u, v)\) in G we add two arcs \((uB, vA)\) and \((vB, uA)\) in D. Finally we set the attribute capacity = 1 for each arc in D [R224].

For a directed graph having \(n\) nodes and \(m\) arcs we derive a directed graph D with \(2n\) nodes and \(m + n\) arcs by replacing each original node \(v\) with two nodes \(vA, vB\) linked by an (internal) arc \((vA, vB)\) in D. Then for each arc \((u, v)\) in G we add one arc \((uB, vA)\) in D. Finally we set the attribute capacity = 1 for each arc in D.

A dictionary with a mapping between nodes in the original graph and the auxiliary digraph is stored as a graph attribute: \(H\).graph[‘mapping’].

**References**

[R224]

4.13 Cores

Find the k-cores of a graph.

The k-core is found by recursively pruning nodes with degrees less than \(k\).

See the following reference for details:

### 4.13.1 core_number

**core_number(G)**

Return the core number for each vertex.

A k-core is a maximal subgraph that contains nodes of degree k or more.

The core number of a node is the largest value k of a k-core containing that node.

**Parameters**

- G : NetworkX graph
  
  A graph or directed graph

**Returns**

- core_number : dictionary
  
  A dictionary keyed by node to the core number.

** Raises **

- NetworkXError

  The k-core is not defined for graphs with self loops or parallel edges.

**Notes**

Not implemented for graphs with parallel edges or self loops.

For directed graphs the node degree is defined to be the in-degree + out-degree.

**References**

[R225]

### 4.13.2 k_core

**k_core(G, k=None, core_number=None)**

Return the k-core of G.

A k-core is a maximal subgraph that contains nodes of degree k or more.

**Parameters**

- G : NetworkX graph
  
  A graph or directed graph

- k : int, optional
  
  The order of the core. If not specified return the main core.

- core_number : dictionary, optional
  
  Precomputed core numbers for the graph G.

**Returns**

- G : NetworkX graph
The k-core subgraph

Raises NetworkXError

The k-core is not defined for graphs with self loops or parallel edges.

See Also:
core_number

Notes

The main core is the core with the largest degree.
Not implemented for graphs with parallel edges or self loops.
For directed graphs the node degree is defined to be the in-degree + out-degree.
Graph, node, and edge attributes are copied to the subgraph.

References

[R226]

4.13.3 k_shell

k_shell (G, k=None, core_number=None)
Return the k-shell of G.

The k-shell is the subgraph of nodes in the k-core but not in the (k+1)-core.

Parameters  G : NetworkX graph
    A graph or directed graph.
  k : int, optional
    The order of the shell. If not specified return the main shell.
  core_number : dictionary, optional
    Precomputed core numbers for the graph G.

Returns  G : NetworkX graph
    The k-shell subgraph

Raises NetworkXError

The k-shell is not defined for graphs with self loops or parallel edges.

See Also:
core_number,k_corona,---------

Shai Carmi, Shlomo Havlin, Scott Kirkpatrick, Yuval Shavitt, and Eran Shir, PNAS July 3, 2007 vol. 104 no. 27 11150-11154

http //www.pnas.org/content/104/27/11150.full
Notes

This is similar to k_corona but in that case only neighbors in the k-core are considered. Not implemented for graphs with parallel edges or self loops. For directed graphs the node degree is defined to be the in-degree + out-degree. Graph, node, and edge attributes are copied to the subgraph.

4.13.4 k_crust

k_crust(\(G, k=\text{None}, \text{core_number}=\text{None}\))

Return the k-crust of \(G\).

The k-crust is the graph \(G\) with the k-core removed.

Parameters

- \(G\) : NetworkX graph
  A graph or directed graph.
- \(k\) : int, optional
  The order of the shell. If not specified return the main crust.
- \(\text{core_number}\) : dictionary, optional
  Precomputed core numbers for the graph \(G\).

Returns

- \(G\) : NetworkX graph
  The k-crust subgraph

Raises

NetworkXError

The k-crust is not defined for graphs with self loops or parallel edges.

See Also:

core_number

Notes

This definition of k-crust is different than the definition in [R228]. The k-crust in [R228] is equivalent to the \(k+1\) crust of this algorithm. Not implemented for graphs with parallel edges or self loops. For directed graphs the node degree is defined to be the in-degree + out-degree. Graph, node, and edge attributes are copied to the subgraph.

References

[R228]
4.13.5 k_corona

**k_corona** *(G, k, core_number=None)*

Return the k-corona of G.

The k-corona is the subgraph of nodes in the k-core which have exactly k neighbours in the k-core.

**Parameters**
- **G**: NetworkX graph
  - A graph or directed graph
- **k**: int
  - The order of the corona.
- **core_number**: dictionary, optional
  - Precomputed core numbers for the graph G.

**Returns**
- **G**: NetworkX graph
  - The k-corona subgraph

**Raises**
- NetworkXError
  - The k-cornoa is not defined for graphs with self loops or parallel edges.

**See Also:**
- core_number

**Notes**

Not implemented for graphs with parallel edges or self loops.

For directed graphs the node degree is defined to be the in-degree + out-degree.

Graph, node, and edge attributes are copied to the subgraph.

**References**

[R227]

4.14 Cycles

**cycle_basis** *(G[, root])*

Returns a list of cycles which form a basis for cycles of G.

**simple_cycles** *(G)*

Find simple cycles (elementary circuits) of a directed graph.

4.14.1 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, optional  

Specify starting node for basis.  

Returns  

A list of cycle lists. Each cycle list is a list of nodes which forms a cycle (loop) in $G$.  

See Also:  

simple_cycles  

Notes  

This is adapted from algorithm CACM 491 [R229].  

References  

[R229]  

Examples  

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

4.14.2 simple_cycles  

simple_cycles ($G$)  

Find simple cycles (elementary circuits) of a directed graph.  

An simple cycle, or elementary circuit, is a closed path where no node appears twice, except that the first and last node are the same. Two elementary circuits are distinct if they are not cyclic permutations of each other.  

This is a nonrecursive, iterator/generator version of Johnson’s algorithm [R230]. There may be better algorithms for some cases [R231][R232].  

Parameters  

$G$ : NetworkX DiGraph  

A directed graph  

Returns  

cycle_generator: generator  

A generator that produces elementary cycles of the graph. Each cycle is a list of nodes with the first and last nodes being the same.  

See Also:  

cycle_basis
Notes

The implementation follows pp. 79-80 in [R230].
The time complexity is \(O((n+e)(c+1))\) for \(n\) nodes, \(e\) edges and \(c\) elementary circuits.
To filter the cycles so that they don’t include certain nodes or edges, copy your graph and eliminate those nodes or edges before calling. >>> copyG = G.copy() >>> copyG.remove_nodes_from([1]) >>> copyG.remove_edges_from(((0,1))) >>> list(nx.simple_cycles(copyG)) [[2], [2, 0], [0]]

References

[R230], [R231], [R232]

Examples

```
>>> G = nx.DiGraph([(0, 0), (0, 1), (0, 2), (1, 2), (2, 0), (2, 1), (2, 2)])
```
```
>>> list(nx.simple_cycles(G))
[[2], [2, 1], [2, 0], [2, 0, 1], [0]]
```

4.15 Directed Acyclic Graphs

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ancestors(G, source)</td>
<td>Return all nodes having a path to \textit{source} in \textit{G}.</td>
</tr>
<tr>
<td>descendants(G, source)</td>
<td>Return all nodes reachable from \textit{source} in \textit{G}.</td>
</tr>
<tr>
<td>topological_sort(G[, nbunch, reverse])</td>
<td>Return a list of nodes in topological sort order.</td>
</tr>
<tr>
<td>topological_sort_recursive(G[, nbunch, reverse])</td>
<td>Return a list of nodes in topological sort order.</td>
</tr>
<tr>
<td>is_directed_acyclic_graph(G)</td>
<td>Return True if the graph \textit{G} is a directed acyclic graph (DAG) or</td>
</tr>
<tr>
<td>is_aperiodic(G)</td>
<td>Return True if \textit{G} is aperiodic.</td>
</tr>
</tbody>
</table>

4.15.1 ancestors

\texttt{ancestors}(G, source)

Return all nodes having a path to \textit{source} in \textit{G}.

\textbf{Parameters}
\begin{itemize}
  \item \texttt{G} : NetworkX DiGraph
  \item \texttt{source} : node in \texttt{G}
\end{itemize}

\textbf{Returns}
\begin{itemize}
  \item \texttt{ancestors} : set()
\end{itemize}

The ancestors of \textit{source} in \textit{G}

4.15.2 descendants

\texttt{descendants}(G, source)

Return all nodes reachable from \textit{source} in \textit{G}.

\textbf{Parameters}
\begin{itemize}
  \item \texttt{G} : NetworkX DiGraph
  \item \texttt{source} : node in \texttt{G}
\end{itemize}

\textbf{Returns}
\begin{itemize}
  \item \texttt{des} : set()
\end{itemize}
The descendants of source in G

4.15.3 topological_sort

`topological_sort(G, nbunch=None, reverse=False)`

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
- `reverse` : bool, optional
  - Return postorder instead of preorder if True. Reverse mode is a bit more efficient.

**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 `NetworkXUnfeasible` exception is raised.

**See Also**
- `is_directed_acyclic_graph`

**Notes**

This algorithm is based on a description and proof in The Algorithm Design Manual [R234].

**References**

[R234]

4.15.4 topological_sort_recursive

`topological_sort_recursive(G, nbunch=None, reverse=False)`

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
- `reverse` : bool, optional
Return postorder instead of preorder if True. Reverse mode is a bit more efficient.

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.

4.15.5 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

4.15.6 is_aperiodic

is_aperiodic(G)

Return True if G is aperiodic.

A directed graph is aperiodic if there is no integer k > 1 that divides the length of every cycle in the graph.

Parameters  G : NetworkX DiGraph

Graph

Returns  aperiodic : boolean

True if the graph is aperiodic False otherwise

Raises NetworkXError

If G is not directed

Notes

This uses the method outlined in [R233], which runs in O(m) time given m edges in G. Note that a graph is not aperiodic if it is acyclic as every integer trivially divides length 0 cycles.
4.16 Distance Measures

Graph diameter, radius, eccentricity and other properties.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>center(G[, e])</td>
<td>Return the center of the graph G.</td>
</tr>
<tr>
<td>diameter(G[, e])</td>
<td>Return the diameter of the graph G.</td>
</tr>
<tr>
<td>eccentricity(G[, v, sp])</td>
<td>Return the eccentricity of nodes in G.</td>
</tr>
<tr>
<td>periphery(G[, e])</td>
<td>Return the periphery of the graph G.</td>
</tr>
<tr>
<td>radius(G[, e])</td>
<td>Return the radius of the graph G.</td>
</tr>
</tbody>
</table>

4.16.1 center

center(G, e=None)
Return the center 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

4.16.2 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
4.16.3 eccentricity

**eccentricity** \((G, v=None, sp=None)\)

Return the eccentricity of nodes in \(G\).

The eccentricity of a node \(v\) is the maximum distance from \(v\) to all other nodes in \(G\).

**Parameters**
- \(G\) : NetworkX graph
  - A graph
- \(v\) : node, optional
  - Return value of specified node
- \(sp\) : dict of dicts, optional
  - All pairs shortest path lengths as a dictionary of dictionaries

**Returns**
- \(ecc\) : dictionary
  - A dictionary of eccentricity values keyed by node.

4.16.4 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

4.16.5 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
4.17 Distance-Regular Graphs

**is_distance_regular(G)**
Returns True if the graph is distance regular, False otherwise.

A connected graph G is distance-regular if for any nodes x,y and any integers i,j=0,1,...,d (where d is the graph diameter), the number of vertices at distance i from x and distance j from y depends only on i,j and the graph distance between x and y, independently of the choice of x and y.

**Parameters**
- G: Networkx graph (undirected)

**Returns**
- bool
  True if the graph is Distance Regular, False otherwise

**See Also:**
intersection_array, global_parameters

**Notes**
For undirected and simple graphs only

**References**
[R237], [R238]

**Examples**
```python
>>> G=nx.hypercube_graph(6)
>>> nx.is_distance_regular(G)
True
```

4.17.2 intersection_array

**intersection_array(G)**
Returns the intersection array of a distance-regular graph.

Given a distance-regular graph G with integers b_i, c_i,i = 0,...,d such that for any 2 vertices x,y in G at a distance i=d(x,y), there are exactly c_i neighbors of y at a distance of i-1 from x and b_i neighbors of y at a distance of i+1 from x.

A distance regular graph’s intersection array is given by, [b_0,b_1,....b_{d-1};c_1,c_2,....c_d]

**Parameters**
- G: Networkx graph (undirected)

**Returns**
- b,c: tuple of lists
See Also:

global_parameters

References

[R236]

Examples

>>> G=nx.icosahedral_graph()
>>> nx.intersection_array(G)
([5, 2, 1], [1, 2, 5])

### 4.17.3 global_parameters

global_parameters(b, c)

Return global parameters for a given intersection array.

Given a distance-regular graph $G$ with integers $b_i$, $c_i$, $i = 0, ..., d$ such that for any 2 vertices $x, y$ in $G$ at a distance $i = d(x, y)$, there are exactly $c_i$ neighbors of $y$ at a distance of $i-1$ from $x$ and $b_i$ neighbors of $y$ at a distance of $i+1$ from $x$.

Thus, a distance-regular graph has the global parameters, $[[c_0, a_0, b_0], [c_1, a_1, b_1], ..., [c_d, a_d, b_d]]$ for the intersection array $[b_0, b_1, ..., b_{d-1}; c_1, c_2, ..., c_d]$ where $a_i + b_i + c_i = k$, $k$ = degree of every vertex.

**Parameters**

- **b, c**: tuple of lists

**Returns**

- **p**: list of three-tuples

See Also:

intersection_array

References

[R235]

Examples

>>> G=nx.dodecahedral_graph()
>>> b,c=nx.intersection_array(G)
>>> list(nx.global_parameters(b,c))
[(0, 0, 3), (1, 0, 2), (1, 1, 1), (1, 1, 1), (2, 0, 1), (3, 0, 0)]

### 4.18 Eulerian

Eulerian circuits and graphs.

<table>
<thead>
<tr>
<th>is_eulerian(G)</th>
<th>Return True if $G$ is an Eulerian graph, False otherwise.</th>
</tr>
</thead>
</table>

Continued on next page
4.18.1 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**

- \(G\) : graph
  - A NetworkX Graph

**Notes**

This implementation requires the graph to be connected (or strongly connected for directed graphs).

**Examples**

```python
>>> 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
```

4.18.2 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 or DiGraph
  - A directed or undirected graph
- \(source\) : node, optional
  - Starting node for circuit.

**Returns**

- \(edges\) : generator
  - A generator that produces edges in the Eulerian circuit.

**Raises**

- NetworkXError
  - If the graph is not Eulerian.

**See Also:**

- is_eulerian

**Notes**

Linear time algorithm, adapted from [R239]. General information about Euler tours [R240].
References

[R239], [R240]

Examples

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

4.19 Flows

4.19.1 Maximum Flow

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>maximum_flow</td>
<td>Find a maximum single-commodity flow.</td>
</tr>
<tr>
<td>maximum_flow_value</td>
<td>Find the value of maximum single-commodity flow.</td>
</tr>
<tr>
<td>minimum_cut</td>
<td>Compute the value and the node partition of a minimum (s, t)-cut.</td>
</tr>
<tr>
<td>minimum_cut_value</td>
<td>Compute the value of a minimum (s, t)-cut.</td>
</tr>
</tbody>
</table>

**maximum_flow**

```python
maximum_flow(G, s, t[, capacity, flow_func])
```

Find 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’.
- `flow_func` : function
  - A function for computing the maximum flow among a pair of nodes in a capacitated graph. The function has to accept at least three parameters: a Graph or Digraph, a source node, and a target node. And return a residual network that follows NetworkX conventions (see Notes). If flow_func is None, the default maximum flow function (`preflow_push()`) is used. See below for alternative algorithms. The choice of...
the default function may change from version to version and should not be relied on. Default value: None.

**kwargs : Any other keyword parameter is passed to the function that computes the maximum flow.

**Returns**

**flow_value** : integer, float
Value of the maximum flow, i.e., net outflow from the source.

**flow_dict** : dict
A dictionary containing the value of the flow that went through each edge.

**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.

**See Also:**
maximum_flow_value(), minimum_cut(), minimum_cut_value(), edmonds_karp(), ford_fulkerson(), preflow_push(), shortest_augmenting_path()

**Notes**
The function used in the flow_func paramter has to return a residual network that follows NetworkX conventions:
The residual network \( R \) from an input graph \( G \) has the same nodes as \( G \). \( R \) is a DiGraph that contains a pair of edges \((u, v)\) and \((v, u)\) iff \((u, v)\) is not a self-loop, and at least one of \((u, v)\) and \((v, u)\) exists in \( G \).

For each edge \((u, v)\) in \( R[u][v][\text{'capacity']}\) is equal to the capacity of \((u, v)\) in \( G \) if it exists in \( G \) or zero otherwise. If the capacity is infinite, \( R[u][v][\text{'capacity']} \) will have a high arbitrary finite value that does not affect the solution of the problem. This value is stored in \( R.graph[\text{'inf']} \). For each edge \((u, v)\) in \( R[u][v][\text{'flow']}\) represents the flow function of \((u, v)\) and satisfies \( R[u][v][\text{'flow']} == -R[v][u][\text{'flow']} \).

The flow value, defined as the total flow into \( t \), the sink, is stored in \( R.graph[\text{'flow_value']} \). Reachability to \( t \) using only edges \((u, v)\) such that \( R[u][v][\text{'flow']} < R[u][v][\text{'capacity']} \) induces a minimum \( s \)-\( t \) cut.

Specific algorithms may store extra data in \( R \).

The function should support an optional boolean parameter value_only. When True, it can optionally terminate the algorithm as soon as the maximum flow value and the minimum cut can be determined.

The legacy ford_fulkerson() maximum flow implementation doesn’t follow this conventions but it is supported as a valid flow_func.

**Examples**
```python
>>> 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)
```

maximum_flow returns both the value of the maximum flow and a dictionary with all flows.

```python
>>> flow_value, flow_dict = nx.maximum_flow(G, 'x', 'y')
>>> flow_value
3.0
>>> print(flow_dict['x']['b'])
1.0
```

You can also use alternative algorithms for computing the maximum flow by using the flow_func parameter.

```python
>>> from networkx.algorithms.flow import shortest_augmenting_path
>>> flow_value == nx.maximum_flow(G, 'x', 'y', flow_func=shortest_augmenting_path)[0]
True
```

### maximum_flow_value

```python
maximum_flow_value(G, s, t, capacity='capacity', flow_func=None, **kwargs)
```

Find the value of 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’.

- **flow_func**: function
  
  A function for computing the maximum flow among a pair of nodes in a capacitated graph. The function has to accept at least three parameters: a Graph or Digraph, a source node, and a target node. And return a residual network that follows NetworkX conventions (see Notes). If flow_func is None, the default maximum flow function (preflow_push()) is used. See below for alternative algorithms. The choice of the default function may change from version to version and should not be relied on. Default value: None.

- **kwargs**: Any other keyword parameter is passed to the function that

---

*NetworkX Reference, Release 1.9*
computes the maximum flow.

Returns **flow_value**: integer, float

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

Raises **NetworkXError**

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

**NetworkXUnbounded**

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

See Also:

maximum_flow(), minimum_cut(), minimum_cut_value(), edmonds_karp(), ford_fulkerson(), preflow_push(), shortest_augmenting_path()

Notes

The function used in the flow_func parameter has to return a residual network that follows NetworkX conventions:

The residual network \( R \) from an input graph \( G \) has the same nodes as \( G \). \( R \) is a DiGraph that contains a pair of edges \((u, v)\) and \((v, u)\) iff \((u, v)\) is not a self-loop, and at least one of \((u, v)\) and \((v, u)\) exists in \( G \).

For each edge \((u, v)\) in \( R \), \( R[u][v]['capacity'] \) is equal to the capacity of \((u, v)\) in \( G \) if it exists in \( G \) or zero otherwise. If the capacity is infinite, \( R[u][v]['capacity'] \) will have a high arbitrary finite value that does not affect the solution of the problem. This value is stored in \( R.graph['inf'] \). For each edge \((u, v)\) in \( R \), \( R[u][v]['flow'] \) represents the flow function of \((u, v)\) and satisfies \( R[u][v]['flow'] = -R[v][u]['flow'] \).

The flow value, defined as the total flow into \( t \), the sink, is stored in \( R.graph['flow_value'] \). Reachability to \( t \) using only edges \((u, v)\) such that \( R[u][v]['flow'] < R[u][v]['capacity'] \) induces a minimum s-t cut.

Specific algorithms may store extra data in \( R \).

The function should support an optional boolean parameter value_only. When True, it can optionally terminate the algorithm as soon as the maximum flow value and the minimum cut can be determined.

The legacy ford_fulkerson() maximum flow implementation doesn’t follow this conventions but it is supported as a valid flow_func.

Examples

```python
>>> 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)
```
maximum_flow_value computes only the value of the maximum flow:

```python
>>> flow_value = nx.maximum_flow_value(G, 'x', 'y')
>>> flow_value
3.0
```

You can also use alternative algorithms for computing the maximum flow by using the flow_func parameter.

```python
>>> from networkx.algorithms.flow import shortest_augmenting_path
>>> flow_value == nx.maximum_flow_value(G, 'x', 'y',
...   flow_func=shortest_augmenting_path)
True
```

**minimum_cut**

`minimum_cut(G, s, t, capacity='capacity', flow_func=None, **kwargs)`

Compute the value and the node partition 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’.

  - **flow_func**: function
    - A function for computing the maximum flow among a pair of nodes in a capacitated graph. The function has to accept at least three parameters: a Graph or Digraph, a source node, and a target node. And return a residual network that follows NetworkX conventions (see Notes). If flow_func is None, the default maximum flow function (preflow_push()) is used. See below for alternative algorithms. The choice of the default function may change from version to version and should not be relied on. Default value: None.

  - **kwargs**: Any other keyword parameter is passed to the function that computes the maximum flow.

**Returns**

- **cut_value**: integer, float
  - Value of the minimum cut.

- **partition**: pair of node sets
  - A partitioning of the nodes that defines a minimum cut.

**Raises**

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

See Also:
maximum_flow(), maximum_flow_value(), minimum_cut_value(), edmonds_karp(), ford_fulkerson(), preflow_push(), shortest_augmenting_path()

Notes

The function used in the flow_func parameter has to return a residual network that follows NetworkX conventions:
The residual network $R$ from an input graph $G$ has the same nodes as $G$. $R$ is a DiGraph that contains a pair of edges $(u, v)$ and $(v, u)$ iff $(u, v)$ is not a self-loop, and at least one of $(u, v)$ and $(v, u)$ exists in $G$.

For each edge $(u, v)$ in $R$, $R[u][v]['capacity']$ is equal to the capacity of $(u, v)$ in $G$ if it exists in $G$ or zero otherwise. If the capacity is infinite, $R[u][v]['capacity']$ will have a high arbitrary finite value that does not affect the solution of the problem. This value is stored in $R$.graph['inf']. For each edge $(u, v)$ in $R$, $R[u][v]['flow']$ represents the flow function of $(u, v)$ and satisfies $R[u][v]['flow'] = -R[v][u]['flow']$.

The flow value, defined as the total flow into $t$, the sink, is stored in $R$.graph['flow_value']. Reachability to $t$ using only edges $(u, v)$ such that $R[u][v]['flow'] < R[u][v]['capacity']$ induces a minimum $s$-$t$ cut.

Specific algorithms may store extra data in $R$.

The function should supports an optional boolean parameter value_only. When True, it can optionally terminate the algorithm as soon as the maximum flow value and the minimum cut can be determined.

Examples

```python
>>> 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)

minimum_cut computes both the value of the minimum cut and the node partition:

```python
>>> cut_value, partition = nx.minimum_cut(G, 'x', 'y')
```

reachable, non_reachable = partition

'partition' here is a tuple with the two sets of nodes that define the minimum cut. You can compute the cut set of edges that induce the minimum cut as follows:

```python
>>> cutset = set()
>>> for u, nbrs in ((n, G[n]) for n in reachable):
...     cutset.update((u, v) for v in nbrs if v in non_reachable)
>>> print(sorted(cutset))
[('c', 'y'), ('x', 'b')]
```
>>> cut_value == sum(G.edge[u][v]['capacity'] for (u, v) in cutset)
True

You can also use alternative algorithms for computing the minimum cut by using the flow_func parameter.

```python
>>> from networkx.algorithms.flow import shortest_augmenting_path
```  
```python
>>> cut_value == nx.minimum_cut(G, 'x', 'y',
...                          flow_func=shortest_augmenting_path)[0]
True
```

**minimum_cut_value**

`minimum_cut_value(G, s, t, capacity='capacity', flow_func=None, **kwargs)`

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’.

- `flow_func`: function
  - A function for computing the maximum flow among a pair of nodes in a capacitated graph. The function has to accept at least three parameters: a Graph or Digraph, a source node, and a target node. And return a residual network that follows NetworkX conventions (see Notes). If flow_func is None, the default maximum flow function `preflow_push()` is used. See below for alternative algorithms. The choice of the default function may change from version to version and should not be relied on. Default value: None.

- `kwargs`: Any other keyword parameter is passed to the function that computes the maximum flow.

**Returns**

- `cut_value`: 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.

**See Also:**

4.19. Flows 245
Notes

The function used in the flow_func parameter has to return a residual network that follows NetworkX conventions:

- The residual network $R$ from an input graph $G$ has the same nodes as $G$. $R$ is a DiGraph that contains a pair of edges $(u, v)$ and $(v, u)$ iff $(u, v)$ is not a self-loop, and at least one of $(u, v)$ and $(v, u)$ exists in $G$.

- For each edge $(u, v)$ in $R$, $R[u][v]['capacity']$ is equal to the capacity of $(u, v)$ in $G$ if it exists in $G$ or zero otherwise. If the capacity is infinite, $R[u][v]['capacity']$ will have a high arbitrary finite value that does not affect the solution of the problem. This value is stored in $R$.graph['inf']. For each edge $(u, v)$ in $R$, $R[u][v]['flow']$ represents the flow function of $(u, v)$ and satisfies $R[u][v]['flow'] = -R[v][u]['flow']$.

- The flow value, defined as the total flow into $t$, the sink, is stored in $R$.graph['flow_value']. Reachability to $t$ using only edges $(u, v)$ such that $R[u][v]['flow'] < R[u][v]['capacity']$ induces a minimum $s$-$t$ cut.

- Specific algorithms may store extra data in $R$.

- The function should supports an optional boolean parameter value_only. When True, it can optionally terminate the algorithm as soon as the maximum flow value and the minimum cut can be determined.

Examples

```python
>>> 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)

minimum_cut_value computes only the value of the minimum cut:

```python
>>> cut_value = nx.minimum_cut_value(G, 'x', 'y')
>>> cut_value
3.0
```

You can also use alternative algorithms for computing the minimum cut by using the flow_func parameter.

```python
>>> from networkx.algorithms.flow import shortest_augmenting_path
>>> cut_value == nx.minimum_cut_value(G, 'x', 'y',
...     flow_func=shortest_augmenting_path)
True
```

### 4.19.2 Edmonds-Karp

...
edmonds_karp(G, s, t[, capacity, residual, ...]) Find a maximum single-commodity flow using the Edmonds-Karp algorithm.

edmonds_karp (G, s, t, capacity='capacity', residual=None, value_only=False, cutoff=None)

Find a maximum single-commodity flow using the Edmonds-Karp algorithm.

This function returns the residual network resulting after computing the maximum flow. See below for details about the conventions NetworkX uses for defining residual networks.

This algorithm has 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'.

- **residual**: NetworkX graph
  Residual network on which the algorithm is to be executed. If None, a new residual network is created. Default value: None.

- **value_only**: bool
  If True compute only the value of the maximum flow. This parameter will be ignored by this algorithm because it is not applicable.

- **cutoff**: integer, float
  If specified, the algorithm will terminate when the flow value reaches or exceeds the cutoff. In this case, it may be unable to immediately determine a minimum cut. Default value: None.

Returns

- **R**: NetworkX DiGraph
  Residual network after computing the maximum flow.

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.
See Also:

- maximum_flow()
- minimum_cut()
- ford_fulkerson()
- preflow_push()
- shortest_augmenting_path()

Notes

The residual network $R$ from an input graph $G$ has the same nodes as $G$. $R$ is a DiGraph that contains a pair of edges $(u, v)$ and $(v, u)$ iff $(u, v)$ is not a self-loop, and at least one of $(u, v)$ and $(v, u)$ exists in $G$.

For each edge $(u, v)$ in $R$, $R[u][v]['capacity']$ is equal to the capacity of $(u, v)$ in $G$ if it exists in $G$ or zero otherwise. If the capacity is infinite, $R[u][v]['capacity']$ will have a high arbitrary finite value that does not affect the solution of the problem. This value is stored in $R$.graph['inf']. For each edge $(u, v)$ in $R$, $R[u][v]['flow']$ represents the flow function of $(u, v)$ and satisfies $R[u][v]['flow'] == -R[v][u]['flow']$.

The flow value, defined as the total flow into $t$, the sink, is stored in $R$.graph['flow_value']. If cutoff is not specified, reachability to $t$ using only edges $(u, v)$ such that $R[u][v]['flow'] < R[u][v]['capacity']$ induces a minimum s-t cut.

Examples

```python
>>> import networkx as nx
>>> from networkx.algorithms.flow import edmonds_karp
```

The functions that implement flow algorithms and output a residual network, such as this one, are not imported to the base NetworkX namespace, so you have to explicitly import them from the flow package.

```python
>>> 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)
>>> R = edmonds_karp(G, 'x', 'y')
>>> flow_value = nx.maximum_flow_value(G, 'x', 'y')
>>> flow_value
3.0
>>> flow_value == R.graph['flow_value']
True
```

4.19.3 Ford-Fulkerson

- ford_fulkerson(G, s, t[, capacity])  Find a maximum single-commodity flow using the Ford-Fulkerson algorithm.
This is the legacy implementation of maximum flow. See Notes below.

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**

- **R** : NetworkX DiGraph
  The residual network after computing the maximum flow. This is a legacy implementation, see Notes and Examples.

**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.

**See Also:**

- maximum_flow()
- minimum_cut()
- edmonds_karp()
- preflow_push()
- shortest_augmenting_path()

**Notes**

This is a legacy implementation of maximum flow (before 1.9). This function used to return a tuple with the flow value and the flow dictionary. Now it returns the residual network resulting after computing the maximum flow, in order to follow the new interface to flow algorithms introduced in NetworkX 1.9.

Note however that the residual network returned by this function does not follow the conventions for residual networks used by the new algorithms introduced in 1.9. This residual network has edges with capacity equal to the capacity of the edge in the original network minus the flow that went through that edge. A dictionary with infinite capacity edges can be found as an attribute of the residual network.

**Examples**

```python
>>> import networkx as nx
>>> from networkx.algorithms.flow import ford_fulkerson
```
The functions that implement flow algorithms and output a residual network, such as this one, are not imported to the base NetworkX namespace, so you have to explicitly import them from the flow package.

```python
>>> 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)
```

This function returns the residual network after computing the maximum flow. This network has graph attributes that contain: a dictionary with edges with infinite capacity flows, the flow value, and a dictionary of flows:

```python
>>> R = ford_fulkerson(G, 'x', 'y')
>>> # A dictionary with infinite capacity flows can be found as an
>>> # attribute of the residual network
>>> inf_capacity_flows = R.graph['inf_capacity_flows']
>>> # There are also attributes for the flow value and the flow dict
>>> flow_value = R.graph['flow_value']
>>> flow_dict = R.graph['flow_dict']
```

You can use the interface to flow algorithms introduced in 1.9 to get the output that the function ford_fulkerson used to produce:

```python
>>> flow_value, flow_dict = nx.maximum_flow(G, 'x', 'y',
... flow_func=ford_fulkerson)
```

### 4.19.4 Shortest Augmenting Path

`shortest_augmenting_path(G, s, t[, ...])` Find a maximum single-commodity flow using the shortest augmenting path algorithm.

#### shortest_augmenting_path

`shortest_augmenting_path(G, s, t, capacity='capacity', residual=None, value_only=False, two_phase=False, cutoff=None)` Find a maximum single-commodity flow using the shortest augmenting path algorithm.

This function returns the residual network resulting after computing the maximum flow. See below for details about the conventions NetworkX uses for defining residual networks.

This algorithm has a running time of \(O(n^2m)\) 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’.

residual : NetworkX graph

Residual network on which the algorithm is to be executed. If None, a new residual network is created. Default value: None.

value_only : bool

If True compute only the value of the maximum flow. This parameter will be ignored by this algorithm because it is not applicable.

two_phase : bool

If True, a two-phase variant is used. The two-phase variant improves the running time on unit-capacity networks from $O(nm)$ to $O(\min(n^{2/3}, m^{1/2})m)$. Default value: False.

cutoff : integer, float

If specified, the algorithm will terminate when the flow value reaches or exceeds the cutoff. In this case, it may be unable to immediately determine a minimum cut. Default value: None.

Returns R : NetworkX DiGraph

Residual network after computing the maximum flow.

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.

See Also:

maximum_flow(), minimum_cut(), edmonds_karp(), ford_fulkerson(), preflow_push()

Notes

The residual network R from an input graph G has the same nodes as G. R is a DiGraph that contains a pair of edges (u, v) and (v, u) iff (u, v) is not a self-loop, and at least one of (u, v) and (v, u) exists in G.

For each edge (u, v) in R, R[u][v][‘capacity’] is equal to the capacity of (u, v) in G if it exists in G or zero otherwise. If the capacity is infinite, R[u][v][‘capacity’] will have a high arbitrary finite value that does not affect the solution of the problem. This value is stored in R.graph[‘inf’]. For each edge (u, v) in R, R[u][v][‘flow’] represents the flow function of (u, v) and satisfies R[u][v][‘flow’] == -R[v][u][‘flow’].

The flow value, defined as the total flow into t, the sink, is stored in R.graph[‘flow_value’]. If cutoff is not specified, reachability to t using only edges (u, v) such that R[u][v][‘flow’] < R[u][v][‘capacity’] induces a minimum s-t cut.
Examples

```python
>>> import networkx as nx
>>> from networkx.algorithms.flow import shortest_augmenting_path

The functions that implement flow algorithms and output a residual network, such as this one, are not imported
to the base NetworkX namespace, so you have to explicitly import them from the flow package.
```
```
>>> 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)
>>> R = shortest_augmenting_path(G, 'x', 'y')
>>> flow_value = nx.maximum_flow_value(G, 'x', 'y')
>>> flow_value
3.0
>>> flow_value == R.graph['flow_value']
True
```

4.19.5 Preflow-Push

`preflow_push(G, s, t[, capacity, residual, ...])` Find a maximum single-commodity flow using the highest-label preflow-push algorithm.

**preflow_push**

`preflow_push (G, s, t, capacity='capacity', residual=None, global_relabel_freq=1, value_only=False)`

Find a maximum single-commodity flow using the highest-label preflow-push algorithm.

This function returns the residual network resulting after computing the maximum flow. See below for details
about the conventions NetworkX uses for defining residual networks.

This algorithm has a running time of $O(n^2 \sqrt{m})$ 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’.
  - **residual**: NetworkX graph
Residual network on which the algorithm is to be executed. If None, a new residual network is created. Default value: None.

global_relabel_freq : integer, float

Relative frequency of applying the global relabeling heuristic to speed up the algorithm. If it is None, the heuristic is disabled. Default value: 1.

value_only : bool

If False, compute a maximum flow; otherwise, compute a maximum preflow which is enough for computing the maximum flow value. Default value: False.

Returns  
R : NetworkX DiGraph

Residual network after computing the maximum flow.

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.

See Also:
maximum_flow(), minimum_cut(), edmonds_karp(), ford_fulkerson(), shortest_augmenting_path()

Notes

The residual network \( R \) from an input graph \( G \) has the same nodes as \( G \). \( R \) is a DiGraph that contains a pair of edges \((u, v)\) and \((v, u)\) iff \((u, v)\) is not a self-loop, and at least one of \((u, v)\) and \((v, u)\) exists in \( G \). For each node \( u \) in \( R \), \( R.node[u]['excess'] \) represents the difference between flow into \( u \) and flow out of \( u \).

For each edge \((u, v)\) in \( R \), \( R[u][v]['capacity'] \) is equal to the capacity of \((u, v)\) in \( G \) if it exists in \( G \) or zero otherwise. If the capacity is infinite, \( R[u][v]['capacity'] \) will have a high arbitrary finite value that does not affect the solution of the problem. This value is stored in \( R.graph['inf'] \). For each edge \((u, v)\) in \( R \), \( R[u][v]['flow'] \) represents the flow function of \((u, v)\) and satisfies \( R[u][v]['flow'] == -R[v][u]['flow'] \).

The flow value, defined as the total flow into \( t \), the sink, is stored in \( R.graph['flow_value'] \). Reachability to \( t \) using only edges \((u, v)\) such that \( R[u][v]['flow'] < R[u][v]['capacity'] \) induces a minimum \( s \)-\( t \) cut.

Examples

```python
>>> import networkx as nx
>>> from networkx.algorithms.flow import preflow_push
```

The functions that implement flow algorithms and output a residual network, such as this one, are not imported to the base NetworkX namespace, so you have to explicitly import them from the flow package.
>>> 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)
>>> R = preflow_push(G, 'x', 'y')
>>> flow_value = nx.maximum_flow_value(G, 'x', 'y')
>>> flow_value == R.graph['flow_value']
True
>>> # preflow_push also stores the maximum flow value
>>> # in the excess attribute of the sink node t
>>> flow_value == R.node['y']['excess']
True
>>> # For some problems, you might only want to compute a
>>> # maximum preflow.
>>> R = preflow_push(G, 'x', 'y', value_only=True)
>>> flow_value == R.graph['flow_value']
True
>>> flow_value == R.node['y']['excess']
True

4.19.6 Utils

build_residual_network(G, capacity)  Build a residual network and initialize a zero flow.

build_residual_network

build_residual_network (G, capacity)

Build a residual network and initialize a zero flow.

The residual network \( R \) from an input graph \( G \) has the same nodes as \( G \). \( R \) is a DiGraph that contains a pair of edges \((u, v)\) and \((v, u)\) iff \((u, v)\) is not a self-loop, and at least one of \((u, v)\) and \((v, u)\) exists in \( G \).

For each edge \((u, v)\) in \( R \), \( R[u][v]['capacity'] \) is equal to the capacity of \((u, v)\) in \( G \) if it exists in \( G \) or zero otherwise. If the capacity is infinite, \( R[u][v]['capacity'] \) will have a high arbitrary finite value that does not affect the solution of the problem. This value is stored in \( R\.graph['inf'] \). For each edge \((u, v)\) in \( R \), \( R[u][v]['flow'] \) represents the flow function of \((u, v)\) and satisfies \( R[u][v]['flow'] == -R[v][u]['flow'] \).

The flow value, defined as the total flow into \( t \), the sink, is stored in \( R\.graph['flow_value'] \). If \( \text{cutoff} \) is not specified, reachability to \( t \) using only edges \((u, v)\) such that \( R[u][v]['flow'] < R[u][v]['capacity'] \) induces a minimum \( s-t \) cut.

4.19.7 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 \( G \).
### Table 4.56 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>min_cost_flow(G[, demand, capacity, weight])</code></td>
<td>Return a minimum cost flow satisfying all demands in digraph G.</td>
</tr>
<tr>
<td><code>cost_of_flow(G, flowDict[, weight])</code></td>
<td>Compute the cost of the flow given by flowDict on graph G.</td>
</tr>
<tr>
<td><code>max_flow_min_cost(G, s, t[, capacity, weight])</code></td>
<td>Return a maximum (s, t)-flow of minimum cost.</td>
</tr>
</tbody>
</table>

#### 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, not connected or is a multigraph.

- **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

**Notes**

This algorithm is not guaranteed to work if edge weights are floating point numbers (overflows and roundoff errors can cause problems).

**References**


**Examples**

A simple example of a min cost flow problem.

```python
>>> 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.

```python
>>> G=nx.DiGraph()
>>> G.add_weighted_edges_from([('s','u',10), ('s','x',5), ...
   ('u','v',1), ('u','x',2), ...
   ('v','y',1), ('x','u',3), ...
   ('x','v',5), ('x','y',2), ...
   ('y','s',7), ('y','v',6)])
>>> G.add_node('s', demand = -1)
>>> G.add_node('v', demand = 1)
>>> flowCost, flowDict = nx.network_simplex(G)
>>> flowCost == nx.shortest_path_length(G, 's', 'v', weight = 'weight')
True
>>> sorted([(u, v) for u in flowDict for v in flowDict[u] if flowDict[u][v] > 0])
[('s', 'x'), ('u', 'v'), ('x', 'u')]
>>> nx.shortest_path(G, 's', 'v', weight = 'weight')
['s', 'x', 'u', 'v']
```
It is possible to change the name of the attributes used for the algorithm.

```python
>>> 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':
```

**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.

```python
>>> 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
```

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.

```python
>>> 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)
```

**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`

**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

```python
>>> G = nx.DiGraph()
>>> G.add_edges_from([(1, 2, {'capacity': 12, 'weight': 4}),
... (1, 3, {'capacity': 20, 'weight': 6}),
... (2, 3, {'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
>>> from networkx.algorithms.flow import ford_fulkerson
>>> R = ford_fulkerson(G, 1, 7)
>>> maxFlow = R.graph['flow_dict']
>>> 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.maximum_flow_value(G, 1, 7)
True
```

### 4.19.8 Capacity Scaling Minimum Cost Flow

`capacity_scaling(G[, demand, capacity, ...])` Find a minimum cost flow satisfying all demands in digraph \( G \).

The function `capacity_scaling` is a capacity scaling successive shortest augmenting path algorithm. Given a digraph \( G \) 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 or MultiDiGraph 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’.

**heap**: class

Type of heap to be used in the algorithm. It should be a subclass of MinHeap or implement a compatible interface.

If a stock heap implementation is to be used, BinaryHeap is recommended over PairingHeap for Python implementations without optimized attribute accesses (e.g., CPython) despite a slower asymptotic running time. For Python implementations with optimized attribute accesses (e.g., PyPy), PairingHeap provides better performance. Default value: BinaryHeap.

Returns

- **flowCost**: integer
  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) if G is a digraph.

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

Raises

- **NetworkXError**
  This exception is raised if the input graph is not directed, not connected or is a multigraph.

- **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:

network_simplex()
Notes

This algorithm does not work if edge weights are floating-point numbers.

Examples

A simple example of a min cost flow problem.

```python
>>> 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.capacity_scaling(G)
>>> flowCost
24
>>> flowDict
{'a': {'c': 1, 'b': 4}, 'c': {'d': 1}, 'b': {'d': 4}, 'd': {}}
```

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

```python
>>> 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('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.capacity_scaling(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':
```

4.20 Graphical degree sequence

Test sequences for graphiness.

- `is_graphical(sequence[, method])` - Returns True if sequence is a valid degree sequence.
- `is_digraphical(in_sequence, out_sequence)` - Returns True if some directed graph can realize the in- and out-degree
- `is_multigraphical(sequence)` - Returns True if some multigraph can realize the sequence.

Continued on next page
4.20.1 is_graphical

is_graphical (sequence, method='eg')
Returns True if sequence is a valid degree sequence.
A degree sequence is valid if some graph can realize it.

Parameters:
- sequence: list or iterable container
  A sequence of integer node degrees
- method: “eg” | “hh”
  The method used to validate the degree sequence. “eg” corresponds to the Erdős-Gallai algorithm, and “hh” to the Havel-Hakimi algorithm.

Returns:
- valid: bool
  True if the sequence is a valid degree sequence and False if not.

References

Erdős-Gallai [EG1960], [choudum1986]
Havel-Hakimi [havel1955], [hakimi1962], [CL1996]

Examples

```python
>>> G = nx.path_graph(4)
>>> sequence = G.degree().values()
>>> nx.is_valid_degree_sequence(sequence)
True
```

4.20.2 is_digraphical

is_digraphical (in_sequence, out_sequence)
Returns True if some directed graph can realize the in- and out-degree sequences.

Parameters:
- in_sequence: list or iterable container
  A sequence of integer node in-degrees
- out_sequence: list or iterable container
  A sequence of integer node out-degrees

Returns:
- valid: bool
  True if in and out-sequences are digraphic False if not.
Notes

This algorithm is from Kleitman and Wang [R241]. The worst case runtime is $O(s \times \log n)$ where $s$ and $n$ are the sum and length of the sequences respectively.

References

[R241]

4.20.3 is_multigraphical

is_multigraphical(\textit{sequence})

Returns True if some multigraph can realize the sequence.

Parameters deg_sequence : list

A list of integers

Returns valid : bool

True if \textit{deg_sequence} is a multigraphic degree sequence and False if not.

Notes

The worst-case run time is $O(n)$ where $n$ is the length of the sequence.

References

[R242]

4.20.4 is_pseudographical

is_pseudographical(\textit{sequence})

Returns True if some pseudograph can realize the sequence.

Every nonnegative integer sequence with an even sum is pseudographical (see [R243]).

Parameters sequence : list or iterable container

A sequence of integer node degrees

Returns valid : bool

True if the \textit{sequence} is a pseudographic degree sequence and False if not.

Notes

The worst-case run time is $O(n)$ where $n$ is the length of the sequence.

References

[R243]
4.20.5 is_valid_degree_sequence_havel_hakimi

is_valid_degree_sequence_havel_hakimi\(\text{(deg\_sequence)}\)

Returns True if \text{deg\_sequence} can be realized by a simple graph.

The validation proceeds using the Havel-Hakimi theorem. Worst-case run time is: \(O(s)\) where \(s\) is the sum of the sequence.

\textbf{Parameters} \text{deg\_sequence} : list

A list of integers where each element specifies the degree of a node in a graph.

\textbf{Returns} \text{valid} : bool

True if \text{deg\_sequence} is graphical and False if not.

\textbf{Notes}

The ZZ condition says that for the sequence \(d\) if

\[
|d| \geq \frac{(\max(d) + \min(d) + 1)^2}{4 \times \min(d)}
\]

then \(d\) is graphical. This was shown in Theorem 6 in [R246].

\textbf{References}

[havel1955], [hakimi1962], [CL1996]

[R246]

4.20.6 is_valid_degree_sequence_erdos_gallai

is_valid_degree_sequence_erdos_gallai\(\text{(deg\_sequence)}\)

Returns True if \text{deg\_sequence} can be realized by a simple graph.

The validation is done using the Erdős-Gallai theorem [EG1960].

\textbf{Parameters} \text{deg\_sequence} : list

A list of integers

\textbf{Returns} \text{valid} : bool

True if \text{deg\_sequence} is graphical and False if not.

\textbf{Notes}

This implementation uses an equivalent form of the Erdős-Gallai criterion. Worst-case run time is: \(O(n)\) where \(n\) is the length of the sequence.

Specifically, a sequence \(d\) is graphical if and only if the sum of the sequence is even and for all strong indices \(k\) in the sequence,
\[
\sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{j=k+1}^{n} \min(d_i, k) = k(n-1) - (k \sum_{j=0}^{k-1} n_j - \sum_{j=0}^{k-1} jn_j)
\]

A strong index \(k\) is any index where \(d_k \geq k\) and the value \(n_j\) is the number of occurrences of \(j\) in \(d\). The maximal strong index is called the Durfee index.

This particular rearrangement comes from the proof of Theorem 3 in [R245].

The ZZ condition says that for the sequence \(d\) if

\[
|d| > \frac{(\max(d) + \min(d) + 1)^2}{4 \times \min(d)}
\]

then \(d\) is graphical. This was shown in Theorem 6 in [R245].

References

[EG1960], [choudum1986]

[R244], [R245]

4.21 Hierarchy

Flow Hierarchy.

flow_hierarchy(G[, weight])  Returns the flow hierarchy of a directed network.

4.21.1 flow_hierarchy

flow_hierarchy (G, weight=None)

Returns the flow hierarchy of a directed network.

Flow hierarchy is defined as the fraction of edges not participating in cycles in a directed graph [R247].

Parameters  G : DiGraph or MultiDiGraph

A directed graph

weight : key,optional (default=None)

Attribute to use for node weights. If None the weight defaults to 1.

Returns  h : float

Flow heirarchy value

Notes

The algorithm described in [R247] computes the flow hierarchy through exponentiation of the adjacency matrix. This function implements an alternative approach that finds strongly connected components. An edge is in a
cycle if and only if it is in a strongly connected component, which can be found in $O(m)$ time using Tarjan’s algorithm.

References

[R247]

4.22 Isolates

Functions for identifying isolate (degree zero) nodes.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>is_isolate(G, n)</code></td>
<td>Determine if node $n$ is an isolate (degree zero).</td>
</tr>
<tr>
<td><code>isolates(G)</code></td>
<td>Return a list of isolates in the graph.</td>
</tr>
</tbody>
</table>

4.22.1 is_isolate

`is_isolate(G, n)`

Determine if node $n$ is an isolate (degree zero).

Parameters

- $G$: graph
  - A networkx graph
- $n$: node
  - A node in $G$

Returns

- `isolate`: bool
  - True if $n$ has no neighbors, False otherwise.

Examples

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

4.22.2 isolates

`isolates(G)`

Return a 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**

```python
g = nx.Graph()
g.add_edge(1, 2)
g.add_node(3)
nx.isolates(g)  
```

To remove all isolates in the graph use
```python
g.remove_nodes_from(nx.isolates(g))
g.nodes()  
```

For digraphs isolates have zero in-degree and zero out_degree
```python
g = nx.DiGraph([(0, 1), (1, 2)])
g.add_node(3)
nx.isolates(g)  
```

### 4.23 Isomorphism

- **is_isomorphic**
  ```python
  is_isomorphic(G1, G2[, node_match, edge_match])
  Returns True if the graphs G1 and G2 are isomorphic and False otherwise.
  ```

- **could_be_isomorphic**
  ```python
  could_be_isomorphic(G1, G2)
  Returns False if graphs are definitely not isomorphic.
  ```

- **fast_could_be_isomorphic**
  ```python
  fast_could_be_isomorphic(G1, G2)
  Returns False if graphs are definitely not isomorphic.
  ```

- **faster_could_be_isomorphic**
  ```python
  faster_could_be_isomorphic(G1, G2)
  Returns False if graphs are definitely not isomorphic.
  ```

### 4.23.1 is_isomorphic

**is_isomorphic**
```python
is_isomorphic(G1, G2, node_match=None, edge_match=None)
Returns True if the graphs G1 and G2 are isomorphic and False otherwise.
```  
**Parameters**

- **G1, G2**: graphs

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

  **node_match** : callable
  
  A function that returns True if node n1 in G1 and n2 in G2 should be considered equal during the isomorphism test. If node_match is not specified then node attributes are not considered.

  The function will be called like

  ```python
  node_match(G1.node[n1], G2.node[n2]).
  ```

  That is, the function will receive the node attribute dictionaries for n1 and n2 as inputs.

  **edge_match** : callable
  
  A function that returns True if the edge attribute dictionary for the pair of nodes (u1, v1) in G1 and (u2, v2) in G2 should be considered equal during the isomorphism test. If edge_match is not specified then edge attributes are not considered.

  The function will be called like

  ```python
  edge_match(G1[u1][v1], G2[u2][v2]).
  ```

  That is, the function will receive the edge attribute dictionaries of the edges under consideration.
See Also:

numerical_node_match, numerical_edge_match, numerical_multiedge_match,
categorical_node_match, categorical_edge_match, categorical_multiedge_match

Notes

Uses the vf2 algorithm [R248].

References

[R248]

Examples

>>> import networkx.algorithms.isomorphism as iso

For digraphs G1 and G2, using ‘weight’ edge attribute (default: 1)

>>> G1 = nx.DiGraph()
>>> G2 = nx.DiGraph()
>>> G1.add_path([1,2,3,4],weight=1)
>>> G2.add_path([10,20,30,40],weight=2)
>>> em = iso.numerical_edge_match('weight', 1)
>>> nx.is_isomorphic(G1, G2) # no weights considered
True
>>> nx.is_isomorphic(G1, G2, edge_match=em) # match weights
False

For multidigraphs G1 and G2, using ‘fill’ node attribute (default: ‘’)

>>> G1 = nx.MultiDiGraph()
>>> G2 = nx.MultiDiGraph()
>>> G1.add_nodes_from([1,2,3],fill='red')
>>> G2.add_nodes_from([10,20,30,40],fill='red')
>>> G1.add_path([1,2,3,4],weight=3, linewidth=2.5)
>>> G2.add_path([10,20,30,40],weight=3)
>>> nm = iso.categorical_node_match('fill', 'red')
>>> nx.is_isomorphic(G1, G2, node_match=nm)
True

For multidigraphs G1 and G2, using ‘weight’ edge attribute (default: 7)

>>> G1.add_edge(1,2, weight=7)
>>> G2.add_edge(10,20)
>>> em = iso.numerical_multiedge_match('weight', 7, rtol=1e-6)
>>> nx.is_isomorphic(G1, G2, edge_match=em)
True

For multigraphs G1 and G2, using ‘weight’ and ‘linewidth’ edge attributes with default values 7 and 2.5. Also using ‘fill’ node attribute with default value ‘red’.

>>> em = iso.numerical_multiedge_match(['weight', 'linewidth'], [7, 2.5])
>>> nm = iso.categorical_node_match('fill', 'red')
>>> nx.is_isomorphic(G1, G2, edge_match=em, node_match=nm)
True
4.23.2 could_be_isomorphic

\texttt{could\_be\_isomorphic}(G1, G2)
Returns False if graphs are definitely not isomorphic. True does NOT guarantee isomorphism.

\textbf{Parameters} \ G1, G2 : graphs
The two graphs G1 and G2 must be the same type.

\textbf{Notes}
Checks for matching degree, triangle, and number of cliques sequences.

4.23.3 fast_could_be_isomorphic

\texttt{fast\_could\_be\_isomorphic}(G1, G2)
Returns False if graphs are definitely not isomorphic.
True does NOT guarantee isomorphism.

\textbf{Parameters} \ G1, G2 : graphs
The two graphs G1 and G2 must be the same type.

\textbf{Notes}
Checks for matching degree and triangle sequences.

4.23.4 faster_could_be_isomorphic

\texttt{faster\_could\_be\_isomorphic}(G1, G2)
Returns False if graphs are definitely not isomorphic.
True does NOT guarantee isomorphism.

\textbf{Parameters} \ G1, G2 : graphs
The two graphs G1 and G2 must be the same type.

\textbf{Notes}
Checks for matching degree sequences.

4.23.5 Advanced Interface to VF2 Algorithm

\textbf{VF2 Algorithm}
An implementation of VF2 algorithm for graph isomorphism testing.
The simplest interface to use this module is to call \texttt{networkx.is_isomorphic()}. 
**Introduction**

The GraphMatcher and DiGraphMatcher are responsible for matching graphs or directed graphs in a predetermined manner. This usually means a check for an isomorphism, though other checks are also possible. For example, a subgraph of one graph can be checked for isomorphism to a second graph.

Matching is done via syntactic feasibility. It is also possible to check for semantic feasibility. Feasibility, then, is defined as the logical AND of the two functions.

To include a semantic check, the (Di)GraphMatcher class should be subclassed, and the semantic_feasibility() function should be redefined. By default, the semantic feasibility function always returns True. The effect of this is that semantics are not considered in the matching of G1 and G2.

**Examples**

Suppose G1 and G2 are isomorphic graphs. Verification is as follows:

```python
>>> from networkx.algorithms import isomorphism
>>> G1 = nx.path_graph(4)
>>> G2 = nx.path_graph(4)
>>> GM = isomorphism.GraphMatcher(G1,G2)
>>> GM.is_isomorphic()
True

GM.mapping stores the isomorphism mapping from G1 to G2.

```python
>>> GM.mapping
{0: 0, 1: 1, 2: 2, 3: 3}
``` 

Suppose G1 and G2 are isomorphic directed graphs. Verification is as follows:

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

DiGM.mapping stores the isomorphism mapping from G1 to G2.

```python
>>> DiGM.mapping
{0: 0, 1: 1, 2: 2, 3: 3}
``` 

**Subgraph Isomorphism**

Graph theory literature can be ambiguous about the meaning of the above statement, and we seek to clarify it now.

In the VF2 literature, a mapping $M$ is said to be a graph-subgraph isomorphism iff $M$ is an isomorphism between $G_2$ and a subgraph of $G_1$. Thus, to say that $G_1$ and $G_2$ are graph-subgraph isomorphic is to say that a subgraph of $G_1$ is isomorphic to $G_2$.

Other literature uses the phrase ‘subgraph isomorphic’ as in ‘$G_1$ does not have a subgraph isomorphic to $G_2$’. Another use is as an in adverb for isomorphic. Thus, to say that $G_1$ and $G_2$ are subgraph isomorphic is to say that a subgraph of $G_1$ is isomorphic to $G_2$.

Finally, the term ‘subgraph’ can have multiple meanings. In this context, ‘subgraph’ always means a ‘node-induced subgraph’. Edge-induced subgraph isomorphisms are not directly supported, but one should be able to perform the
check by making use of nx.line_graph(). For subgraphs which are not induced, the term ‘monomorphism’ is preferred over ‘isomorphism’. Currently, it is not possible to check for monomorphisms.

Let $G=(N,E)$ be a graph with a set of nodes $N$ and set of edges $E$.

If $G'=(N',E')$ is a subgraph, then: $N'$ is a subset of $N$ $E'$ is a subset of $E$

If $G'=(N',E')$ is a node-induced subgraph, then: $N'$ is a subset of $N$ $E'$ is the subset of edges in $E$ relating nodes in $N'$

If $G'=(N',E')$ is an edge-induced subgraph, then: $N'$ is the subset of nodes in $N$ related by edges in $E'$ $E'$ is a subset of $E$

References


See Also

syntactic_feasibility(), semantic_feasibility()

Notes

Modified to handle undirected graphs. Modified to handle multiple edges.

In general, this problem is NP-Complete.

Graph Matcher

GraphMatcher.__init__(G1, G2[, node_match, ...]) Initialize graph matcher.

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 mapping G1_node to G2_node is semantically feasible.

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

__init__

GraphMatcher.__init__(G1, G2, node_match=None, edge_match=None) Initialize graph matcher.

Parameters  G1, G2: graph

The graphs to be tested.

4.23. Isomorphism 273
Node match: callable

A function that returns True iff node n1 in G1 and n2 in G2 should be considered equal during the isomorphism test. The function will be called like:

```
node_match(G1.node[n1], G2.node[n2])
```

That is, the function will receive the node attribute dictionaries of the nodes under consideration. If None, then no attributes are considered when testing for an isomorphism.

Edge match: callable

A function that returns True iff the edge attribute dictionary for the pair of nodes (u1, v1) in G1 and (u2, v2) in G2 should be considered equal during the isomorphism test. The function will be called like:

```
edge_match(G1[u1][v1], G2[u2][v2])
```

That is, the function will receive the edge attribute dictionaries of the edges under consideration. If None, then no attributes are considered when testing for an isomorphism.

Initialize

```
GraphMatcher.initialize()
```

Reinitializes the state of the algorithm.

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

Is isomorphic

```
GraphMatcher.is_isomorphic()
```

Returns True if G1 and G2 are isomorphic graphs.

Subgraph is isomorphic

```
GraphMatcher.subgraph_is_isomorphic()
```

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

Isomorphisms iter

```
GraphMatcher.isomorphisms_iter()
```

Generator over isomorphisms between G1 and G2.

Subgraph isomorphisms iter

```
GraphMatcher.subgraph_isomorphisms_iter()
```

Generator over isomorphisms between a subgraph of G1 and G2.

Candidate pairs iter

```
GraphMatcher.candidate_pairs_iter()
```

Iterator over candidate pairs of nodes in G1 and G2.
match
GraphMatcher.match()
   Extends the isomorphism mapping.

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

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

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

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

DiGraph Matcher

DiGraphMatcher.__init__(G1, G2[, ...]) Initialize graph matcher.
   Parameters G1, G2 : graph
      The graphs to be tested.

   node_match : callable
      A function that returns True iff node n1 in G1 and n2 in G2 should be considered equal
during the isomorphism test. The function will be called like:

      node_match(G1.node[n1], G2.node[n2])

      That is, the function will receive the node attribute dictionaries of the nodes under con-
sideration. If None, then no attributes are considered when testing for an isomorphism.

   edge_match : callable
      A function that returns True iff the edge attribute dictionary for the pair of nodes (u1, v1) in G1 and (u2, v2) in G2 should be considered equal during the isomorphism test.
The function will be called like:

    edge_match(G1[u1][v1], G2[u2][v2])

That is, the function will receive the edge attribute dictionaries of the edges under consideration. If None, then no attributes are considered when testing for an isomorphism.

`initialize`

```
DiGraphMatcher.initialize()
```

Reinitializes the state of the algorithm.

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

`is_isomorphic`

```
DiGraphMatcher.is_isomorphic()
```

Returns True if G1 and G2 are isomorphic graphs.

`subgraph_is_isomorphic`

```
DiGraphMatcher.subgraph_is_isomorphic()
```

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

`isomorphisms_iter`

```
DiGraphMatcher.isomorphisms_iter()
```

Generator over isomorphisms between G1 and G2.

`subgraph_isomorphisms_iter`

```
DiGraphMatcher.subgraph_isomorphisms_iter()
```

Generator over isomorphisms between a subgraph of G1 and G2.

`candidate_pairs_iter`

```
DiGraphMatcher.candidate_pairs_iter()
```

Iterator over candidate pairs of nodes in G1 and G2.

`match`

```
DiGraphMatcher.match()
```

Extends the isomorphism mapping.

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

`semantic_feasibility`

```
DiGraphMatcher.semantic_feasibility(G1_node, G2_node)
```

Returns True if mapping G1_node to G2_node is semantically feasible.
syntactic_feasibility

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

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

Match helpers

categorical_node_match(attr, default)
Returns a comparison function for a categorical node attribute.

Parameters

attr: string | list
The categorical node attribute to compare, or a list of categorical node attributes to compare.

default: value | list
The default value for the categorical node attribute, or a list of default values for the categorical node attributes.

Returns

match: function
The customized, categorical node match function.

Examples

>>> import networkx.algorithms.isomorphism as iso
>>> nm = iso.categorical_node_match('size', 1)
>>> nm = iso.categorical_node_match(['color', 'size'], ['red', 2])

categorical_edge_match

categorical_edge_match(attr, default)
Returns a comparison function for a categorical edge attribute.

Parameters

attr: string | list
The categorical edge attribute to compare, or a list of categorical edge attributes to compare.

default: value | list
The default value for the categorical edge attribute, or a list of default values for the categorical edge attributes.

Returns

match: function
The customized, categorical edge match function.

Examples

>>> import networkx.algorithms.isomorphism as iso
>>> nm = iso.categorical_edge_match('color', 'red')
>>> nm = iso.categorical_edge_match(['color', 'size'], ['red', 2])

categorical_multiedge_match

categorical_multiedge_match(attr, default)
Returns a comparison function for a categorical edge attribute.

Parameters

attr: string | list
The categorical multiedge attribute to compare, or a list of categorical multiedge attributes to compare.

default: value | list
The default value for the categorical multiedge attribute, or a list of default values for the categorical multiedge attributes.

Returns

match: function
The customized, categorical multiedge match function.

Examples

>>> import networkx.algorithms.isomorphism as iso
>>> nm = iso.categorical_multiedge_match('color', 'red')
>>> nm = iso.categorical_multiedge_match(['color', 'size'], ['red', 2])

categorical_node_match

categorical_node_match(attr, default)
Returns a comparison function for a categorical node attribute.

Parameters

attr: string | list
The categorical node attribute to compare, or a list of categorical node attributes to compare.

default: value | list
The default value for the categorical node attribute, or a list of default values for the categorical node attributes.

Returns

match: function
The customized, categorical node match function.

Examples

>>> import networkx.algorithms.isomorphism as iso
>>> nm = iso.categorical_node_match('size', 1)
>>> nm = iso.categorical_node_match(['color', 'size'], ['red', 2])

categorical_edge_match

categorical_edge_match(attr, default)
Returns a comparison function for a categorical edge attribute.

Parameters

attr: string | list
The categorical edge attribute to compare, or a list of categorical edge attributes to compare.

default: value | list
The default value for the categorical edge attribute, or a list of default values for the categorical edge attributes.

Returns

match: function
The customized, categorical edge match function.

Examples

>>> import networkx.algorithms.isomorphism as iso
>>> nm = iso.categorical_edge_match('color', 'red')
>>> nm = iso.categorical_edge_match(['color', 'size'], ['red', 2])

categorical_multiedge_match

categorical_multiedge_match(attr, default)
Returns a comparison function for a categorical edge attribute.

Parameters

attr: string | list
The categorical multiedge attribute to compare, or a list of categorical multiedge attributes to compare.

default: value | list
The default value for the categorical multiedge attribute, or a list of default values for the categorical multiedge attributes.

Returns

match: function
The customized, categorical multiedge match function.

Examples

>>> import networkx.algorithms.isomorphism as iso
>>> nm = iso.categorical_multiedge_match('color', 'red')
>>> nm = iso.categorical_multiedge_match(['color', 'size'], ['red', 2])

4.23. Isomorphism
The categorical edge attribute to compare, or a list of categorical edge attributes to compare.

**default**: value | list

The default value for the categorical edge attribute, or a list of default values for the categorical edge attributes.

**Returns** **match**: function

The customized, categorical \textit{edge\_match} function.

**Examples**

```python
>>> import networkx.algorithms.isomorphism as iso
>>> nm = iso.categorical_edge_match('size', 1)
>>> nm = iso.categorical_edge_match(['color', 'size'], ['red', 2])
```

categorical\_multiedge\_match

categorical\_multiedge\_match \textit{(attr, default)}

Returns a comparison function for a categorical edge attribute.

The value(s) of the \textit{attr(s)} must be hashable and comparable via the == operator since they are placed into a set([]) object. If the sets from G1 and G2 are the same, then the constructed function returns True.

**Parameters** \textit{attr}: string | list

The categorical edge attribute to compare, or a list of categorical edge attributes to compare.

\textit{default}: value | list

The default value for the categorical edge attribute, or a list of default values for the categorical edge attributes.

**Returns** **match**: function

The customized, categorical \textit{edge\_match} function.

**Examples**

```python
>>> import networkx.algorithms.isomorphism as iso
>>> nm = iso.categorical_multiedge_match('size', 1)
>>> nm = iso.categorical_multiedge_match(['color', 'size'], ['red', 2])
```

categorical\_multiedge\_match

categorical\_multiedge\_match \textit{(attr, default)}

Returns a comparison function for a categorical edge attribute.

The value(s) of the \textit{attr(s)} must be hashable and comparable via the == operator since they are placed into a set([]) object. If the sets from G1 and G2 are the same, then the constructed function returns True.

**Parameters** \textit{attr}: string | list

The categorical edge attribute to compare, or a list of categorical edge attributes to compare.

\textit{default}: value | list

The default value for the categorical edge attribute, or a list of default values for the categorical edge attributes.

**Returns** **match**: function

The customized, categorical \textit{edge\_match} function.

**Examples**

```python
>>> import networkx.algorithms.isomorphism as iso
>>> nm = iso.categorical_multiedge_match('size', 1)
>>> nm = iso.categorical_multiedge_match(['color', 'size'], ['red', 2])
```

numerical\_node\_match

numerical\_node\_match \textit{(attr, default, rtol=1e-05, atol=1e-08)}

Returns a comparison function for a numerical node attribute.

The value(s) of the \textit{attr(s)} must be numerical and sortable. If the sorted list of values from G1 and G2 are the same within some tolerance, then the constructed function returns True.

**Parameters** \textit{attr}: string | list

The numerical node attribute to compare, or a list of numerical node attributes to compare.

\textit{default}: value | list

The default value for the numerical node attribute, or a list of default values for the numerical node attributes.

**Examples**

```python
>>> import networkx.algorithms.isomorphism as iso
>>> nm = iso.categorical_multiedge_match('size', 1)
>>> nm = iso.categorical_multiedge_match(['color', 'size'], ['red', 2])
```
The default value for the numerical node attribute, or a list of default values for the numerical node attributes.

rtol : float
The relative error tolerance.

atol : float
The absolute error tolerance.

Returns match : function
The customized, numerical node match function.

Examples

```python
>>> import networkx.algorithms.isomorphism as iso
>>> nm = iso.numerical_node_match('weight', 1.0)
>>> nm = iso.numerical_node_match(['weight', 'linewidth'], [.25, .5])
```

e numerical_edge_match
e numerical_edge_match (attr, default, rtol=1e-05, atol=1e-08)
Returns a comparison function for a numerical edge attribute.

The value(s) of the attr(s) must be numerical and sortable. If the sorted list of values from G1 and G2 are the same within some tolerance, then the constructed function returns True.

Parameters attr : string | list
The numerical edge attribute to compare, or a list of numerical edge attributes to compare.

default : value | list
The default value for the numerical edge attribute, or a list of default values for the numerical edge attributes.

rtol : float
The relative error tolerance.

atol : float
The absolute error tolerance.

Returns match : function
The customized, numerical edge match function.

Examples

```python
>>> import networkx.algorithms.isomorphism as iso
>>> nm = iso.numerical_edge_match('weight', 1.0)
>>> nm = iso.numerical_edge_match(['weight', 'linewidth'], [.25, .5])
```
**numerical_multiedge_match**

**numerical_multiedge_match**(*attr*, *default*, *rtol=1e-05*, *atol=1e-08*)

Returns a comparison function for a numerical edge attribute.

The value(s) of the attr(s) must be numerical and sortable. If the sorted list of values from G1 and G2 are the same within some tolerance, then the constructed function returns True.

**Parameters**

*attr*: string | list

The numerical edge attribute to compare, or a list of numerical edge attributes to compare.

*default*: value | list

The default value for the numerical edge attribute, or a list of default values for the numerical edge attributes.

*rtol*: float

The relative error tolerance.

*atol*: float

The absolute error tolerance.

**Returns**

*match*: function

The customized, numerical edge match function.

**Examples**

```python
>>> import networkx.algorithms.isomorphism as iso
>>> nm = iso.numerical_multiedge_match('weight', 1.0)
>>> nm = iso.numerical_multiedge_match(['weight', 'linewidth'], [.25, .5])
```

**generic_node_match**

**generic_node_match**(*attr*, *default*, *op*)

Returns a comparison function for a generic attribute.

The value(s) of the attr(s) are compared using the specified operators. If all the attributes are equal, then the constructed function returns True.

**Parameters**

*attr*: string | list

The node attribute to compare, or a list of node attributes to compare.

*default*: value | list

The default value for the node attribute, or a list of default values for the node attributes.

*op*: callable | list

The operator to use when comparing attribute values, or a list of operators to use when comparing values for each attribute.

**Returns**

*match*: function

The customized, generic node match function.
Examples

```python
>>> from operator import eq
>>> from networkx.algorithms.isomorphism.matchhelpers import close
>>> from networkx.algorithms.isomorphism import generic_node_match
>>> nm = generic_node_match('weight', 1.0, close)
>>> nm = generic_node_match('color', 'red', eq)
>>> nm = generic_node_match(['weight', 'color'], [1.0, 'red'], [close, eq])
```

generic_edge_match

generic_edge_match (attr, default, op)

Returns a comparison function for a generic attribute.

The value(s) of the attr(s) are compared using the specified operators. If all the attributes are equal, then the constructed function returns True.

Parameters

attr : string | list

The edge attribute to compare, or a list of edge attributes to compare.

default : value | list

The default value for the edge attribute, or a list of default values for the edge attributes.

op : callable | list

The operator to use when comparing attribute values, or a list of operators to use when comparing values for each attribute.

Returns

match : function

The customized, generic edge_match function.

Examples

```python
>>> from operator import eq
>>> from networkx.algorithms.isomorphism.matchhelpers import close
>>> from networkx.algorithms.isomorphism import generic_edge_match
>>> nm = generic_edge_match('weight', 1.0, close)
>>> nm = generic_edge_match('color', 'red', eq)
>>> nm = generic_edge_match(['weight', 'color'], [1.0, 'red'], [close, eq])
```

generic_multiedge_match

generic_multiedge_match (attr, default, op)

Returns a comparison function for a generic attribute.

The value(s) of the attr(s) are compared using the specified operators. If all the attributes are equal, then the constructed function returns True. Potentially, the constructed edge_match function can be slow since it must verify that no isomorphism exists between the multiedges before it returns False.

Parameters

attr : string | list

The edge attribute to compare, or a list of node attributes to compare.

default : value | list

The default value for the edge attribute, or a list of default values for the edge attributes.

op : callable | list

The operator to use when comparing attribute values, or a list of operators to use when comparing values for each attribute.
The operator to use when comparing attribute values, or a list of operators to use when comparing values for each attribute.

**Returns**  **match** : function

The customized, generic $edge_{match}$ function.

**Examples**

```python
>>> from operator import eq
>>> from networkx.algorithms.isomorphism.matchhelpers import close
>>> from networkx.algorithms.isomorphism import generic_node_match

>>> nm = generic_node_match('weight', 1.0, close)
>>> nm = generic_node_match('color', 'red', eq)
>>> nm = generic_node_match(['weight', 'color'],
... [1.0, 'red'],
... [close, eq])
```

## 4.24 Link Analysis

### 4.24.1 PageRank

PageRank analysis of graph structure.

```python
p pagerank(G[, alpha, personalization, ...]) Return the PageRank of the nodes in the graph.
p pagerank_numpy(G[, alpha, personalization, ...]) Return the PageRank of the nodes in the graph.
p pagerank_scipy(G[, alpha, personalization, ...]) Return the PageRank of the nodes in the graph.
p google_matrix(G[, alpha, personalization, ...]) Return the Google matrix of the graph.
```

**pagerank**

**pagerank** $(G, \text{alpha}=0.85, \text{personalization}=\text{None}, \text{max_iter}=100, \text{tol}=1e-06, \text{nstart}=\text{None}, \text{weight}='weight', \text{dangling}=\text{None})$

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. Undirected graphs will be converted to a directed graph with two directed edges for each undirected edge.

**alpha** : float, optional

Damping parameter for PageRank, default=0.85.

**personalization** : dict, optional

The “personalization vector” consisting of a dictionary with a key for every graph node and nonzero personalization value for each node. By default, a uniform distribution is used.

**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.

weight : key, optional
   Edge data key to use as weight. If None weights are set to 1.

dangling: dict, optional
   The outedges to be assigned to any “dangling” nodes, i.e., nodes without any outedges. The dict key is the node the outedge points to and the dict value is the weight of that outedge. By default, dangling nodes are given outedges according to the personalization vector (uniform if not specified). This must be selected to result in an irreducible transition matrix (see notes under google_matrix). It may be common to have the dangling dict to be the same as the personalization dict.

Returns pagerank : dictionary
   Dictionary of nodes with PageRank as value

See Also:
pagerank_numpy, pagerank_scipy, google_matrix

Notes

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

The PageRank algorithm was designed for directed graphs but this algorithm does not check if the input graph is directed and will execute on undirected graphs by converting each edge in the directed graph to two edges.

References

[R255], [R256]

Examples

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

pagerank_numpy

 pagerank_numpy (G, alpha=0.85, personalization=None, weight='weight', dangling=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

4.24. Link Analysis
A NetworkX graph. Undirected graphs will be converted to a directed graph with two directed edges for each undirected edge.

**alpha**: float, optional

Damping parameter for PageRank, default=0.85.

**personalization**: dict, optional

The “personalization vector” consisting of a dictionary with a key for every graph node and nonzero personalization value for each node. By default, a uniform distribution is used.

**weight**: key, optional

Edge data key to use as weight. If None weights are set to 1.

**dangling**: dict, optional

The outedges to be assigned to any “dangling” nodes, i.e., nodes without any outedges. The dict key is the node the outedge points to and the dict value is the weight of that outedge. By default, dangling nodes are given outedges according to the personalization vector (uniform if not specified) This must be selected to result in an irreducible transition matrix (see notes under google_matrix). It may be common to have the dangling dict to be the same as the personalization dict.

Returns **pagerank**: dictionary

Dictionary of nodes with PageRank as value.

See Also:

`pagerank`, `pagerank_scipy`, `google_matrix`

Notes

The eigenvector calculation uses NumPy’s interface to the LAPACK eigenvalue solvers. This will be the fastest and most accurate for small graphs.

This implementation works with Multi(Di)Graphs. For multigraphs the weight between two nodes is set to be the sum of all edge weights between those nodes.

References

[R257], [R258]

Examples

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

**pagerank_scipy**

`pagerank_scipy` \((G, alpha=0.85, personalization=None, max_iter=100, tol=1e-06, weight='weight', dangling=None)\)

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. Undirected graphs will be converted to a directed graph with two directed edges for each undirected edge.

- **alpha**: float, optional
  - Damping parameter for PageRank, default=0.85.

- **personalization**: dict, optional
  - The "personalization vector" consisting of a dictionary with a key for every graph node and nonzero personalization value for each node. By default, a uniform distribution is used.

- **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.

- **weight**: key, optional
  - Edge data key to use as weight. If None weights are set to 1.

- **dangling**: dict, optional
  - The outedges to be assigned to any "dangling" nodes, i.e., nodes without any outedges. The dict key is the node the outedge points to and the dict value is the weight of that outedge. By default, dangling nodes are given outedges according to the personalization vector (uniform if not specified) This must be selected to result in an irreducible transition matrix (see notes under google_matrix). It may be common to have the dangling dict to be the same as the personalization dict.

**Returns**

- **pagerank**: dictionary
  - Dictionary of nodes with PageRank as value

See Also:

- pagerank, pagerank_numpy, google_matrix

**Notes**

The eigenvector calculation uses power iteration with a SciPy sparse matrix representation.

This implementation works with Multi(Di)Graphs. For multigraphs the weight between two nodes is set to be the sum of all edge weights between those nodes.

**References**

[R259], [R260]
Examples

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

**google_matrix**

```python
google_matrix(G, alpha=0.85, personalization=None, nodelist=None, weight='weight', dangling=None)
```

Return the Google matrix of the graph.

**Parameters**

- **G**: graph
  - A NetworkX graph. Undirected graphs will be converted to a directed graph with two directed edges for each undirected edge.
- **alpha**: float
  - The damping factor.
- **personalization**: dict, optional
  - The "personalization vector" consisting of a dictionary with a key for every graph node and nonzero personalization value for each node. By default, a uniform distribution is used.
- **nodelist**: list, optional
  - The rows and columns are ordered according to the nodes in nodelist. If nodelist is None, then the ordering is produced by G.nodes().
- **weight**: key, optional
  - Edge data key to use as weight. If None weights are set to 1.
- **dangling**: dict, optional
  - The outedges to be assigned to any “dangling” nodes, i.e., nodes without any outedges. The dict key is the node the outedge points to and the dict value is the weight of that outedge. By default, dangling nodes are given outedges according to the personalization vector (uniform if not specified) This must be selected to result in an irreducible transition matrix (see notes below). It may be common to have the dangling dict to be the same as the personalization dict.

**Returns**

- **A**: NumPy matrix
  - Google matrix of the graph

**See Also:**

- pagerank, pagerank_numpy, pagerank_scipy

**Notes**

The matrix returned represents the transition matrix that describes the Markov chain used in PageRank. For PageRank to converge to a unique solution (i.e., a unique stationary distribution in a Markov chain), the transition matrix must be irreducible. In other words, it must be that there exists a path between every pair of nodes in the graph, or else there is the potential of “rank sinks.”
This implementation works with Multi(Di)Graphs. For multigraphs the weight between two nodes is set to be the sum of all edge weights between those nodes.

### 4.24.2 Hits

Hubs and authorities analysis of graph structure.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hits(G[, max_iter, tol, nstart, normalized])</td>
<td>Return HITS hubs and authorities values for nodes.</td>
</tr>
<tr>
<td>hits_numpy(G[, normalized])</td>
<td>Return HITS hubs and authorities values for nodes.</td>
</tr>
<tr>
<td>hits_scipy(G[, max_iter, tol, normalized])</td>
<td>Return HITS hubs and authorities values for nodes.</td>
</tr>
<tr>
<td>hub_matrix(G[, nodelist])</td>
<td>Return the HITS hub matrix.</td>
</tr>
<tr>
<td>authority_matrix(G[, nodelist])</td>
<td>Return the HITS authority matrix.</td>
</tr>
</tbody>
</table>

**hits**

`hits (G, max_iter=100, tol=1e-08, nstart=None, normalized=True)`  
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** : integer, 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.

- **normalized** : bool (default=True)  
  Normalize results by the sum of all of the values.

**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**

[R249], [R250]
Examples

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

hits_numpy

**hits_numpy** *(G, normalized=True)*

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
- **normalized**: bool (default=True)
  - Normalize results by the sum of all of the values.

**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

[R251], [R252]

Examples

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

hits_scipy

**hits_scipy** *(G, max_iter=100, tol=1e-06, normalized=True)*

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**: integer, optional
- **tol**: float, optional
- **normalized**: bool (default=True)
  - Normalize results by the sum of all of the values.

**Returns**

- **hubs,authorities**: two-tuple of dictionaries
  - Two dictionaries keyed by node containing the hub and authority values.
Maximum number of iterations in power method.

\textbf{tol} : float, optional
Error tolerance used to check convergence in power method iteration.

\textbf{nstart} : dictionary, optional
Starting value of each node for power method iteration.

\textbf{normalized} : bool (default=True)
Normalize results by the sum of all of the values.

\textbf{Returns} (\textbf{hubs,authorities}) : two-tuple of dictionaries
Two dictionaries keyed by node containing the hub and authority values.

\textbf{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)\times 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.

\textbf{References}
[R253], [R254]

\textbf{Examples}

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

\textbf{hub\_matrix}

\textbf{hub\_matrix}(G, nodelist=None)
Return the HITS hub matrix.

\textbf{authority\_matrix}

\textbf{authority\_matrix}(G, nodelist=None)
Return the HITS authority matrix.

\section{4.25 Link Prediction}
Link prediction algorithms.

\textbf{resource\_allocation\_index}(G[, ebunch])
Compute the resource allocation index of all node pairs in ebunch.
Table 4.67 – continued from previous page

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jaccard_coefficient(G[, ebunch])</td>
<td>Compute the Jaccard coefficient of all node pairs in ebunch.</td>
</tr>
<tr>
<td>adamic_adar_index(G[, ebunch])</td>
<td>Compute the Adamic-Adar index of all node pairs in ebunch.</td>
</tr>
<tr>
<td>preferential_attachment(G[, ebunch])</td>
<td>Compute the preferential attachment score of all node pairs in ebunch.</td>
</tr>
<tr>
<td>cn_soundarajan_hopcroft(G[, ebunch, community])</td>
<td>Count the number of common neighbors of all node pairs in ebunch using community information.</td>
</tr>
<tr>
<td>ra_index_soundarajan_hopcroft(G[, ebunch, ...])</td>
<td>Compute the resource allocation index of all node pairs in ebunch using community information.</td>
</tr>
<tr>
<td>within_inter_cluster(G[, ebunch, delta, ...])</td>
<td>Compute the ratio of within- and inter-cluster common neighbors of all node pairs in ebunch.</td>
</tr>
</tbody>
</table>

4.25.1 resource_allocation_index

resource_allocation_index(G, ebunch=None)

   Compute the resource allocation index of all node pairs in ebunch.

   Resource allocation index of u and v is defined as
   \[
   \sum_{w \in \Gamma(u) \cap \Gamma(v)} \frac{1}{|\Gamma(w)|}
   \]

   where \(\Gamma(u)\) denotes the set of neighbors of u.

Parameters

- **G** : graph
  A NetworkX undirected graph.

- **ebunch** : iterable of node pairs, optional (default = None)
  Resource allocation index will be computed for each pair of nodes given in the iterable.
  The pairs must be given as 2-tuples (u, v) where u and v are nodes in the graph. If
  ebunch is None then all non-existent edges in the graph will be used. Default value:
  None.

Returns

- **piter** : iterator
  An iterator of 3-tuples in the form (u, v, p) where (u, v) is a pair of nodes and p is their
  resource allocation index.

References

[R266]

Examples

```python
>>> import networkx as nx
>>> G = nx.complete_graph(5)
>>> preds = nx.resource_allocation_index(G, [(0, 1), (2, 3)])
>>> for u, v, p in preds:
...    print("('(u, v, p")
...    print("%d, %d) -> %.8f" % (u, v, p))
...    print(\n' (0, 1) -> 0.75000000\n' (2, 3) -> 0.75000000')
```
4.25.2 jaccard_coefficient

**jaccard_coefficient** *(G, ebunch=None)*

Compute the Jaccard coefficient of all node pairs in ebunch.

Jaccard coefficient of nodes *u* and *v* is defined as

\[ \frac{|\Gamma(u) \cap \Gamma(v)|}{|
\Gamma(u) \cup \Gamma(v)|} \]

where \( \Gamma(u) \) denotes the set of neighbors of *u*.

**Parameters**

- **G**: graph
  
  A NetworkX undirected graph.

- **ebunch**: iterable of node pairs, optional (default = None)

  Jaccard coefficient will be computed for each pair of nodes given in the iterable. The pairs must be given as 2-tuples (u, v) where u and v are nodes in the graph. If ebunch is None then all non-existent edges in the graph will be used. Default value: None.

**Returns**

- **piter**: iterator

  An iterator of 3-tuples in the form (u, v, p) where (u, v) is a pair of nodes and p is their Jaccard coefficient.

**References**

[R263]

**Examples**

```python
>>> import networkx as nx
>>> G = nx.complete_graph(5)
>>> preds = nx.jaccard_coefficient(G, [(0, 1), (2, 3)])
>>> for u, v, p in preds:
...     print('%d, %d -> %.8f' % (u, v, p))
...     '
...     (0, 1) -> 0.60000000
...     (2, 3) -> 0.60000000
```

4.25.3 adamic_adar_index

**adamic_adar_index** *(G, ebunch=None)*

Compute the Adamic-Adar index of all node pairs in ebunch.

Adamic-Adar index of *u* and *v* is defined as

\[ \sum_{w \in \Gamma(u) \cap \Gamma(v)} \frac{1}{\log |
\Gamma(w)|} \]

where \( \Gamma(u) \) denotes the set of neighbors of *u*.

**Parameters**

- **G**: graph

  NetworkX undirected graph.
**NetworkX Reference, Release 1.9**

**Adamic-Adar Index**

**ebunch**: iterable of node pairs, optional (default = None)

Adamic-Adar index will be computed for each pair of nodes given in the iterable. The pairs must be given as 2-tuples (u, v) where u and v are nodes in the graph. If ebunch is None then all non-existent edges in the graph will be used. Default value: None.

**Returns** piter : iterator

An iterator of 3-tuples in the form (u, v, p) where (u, v) is a pair of nodes and p is their Adamic-Adar index.

**References**

[R261]

**Examples**

```python
>>> import networkx as nx
>>> G = nx.complete_graph(5)
>>> preds = nx.adamic_adar_index(G, [(0, 1), (2, 3)])
>>> for u, v, p in preds:
...     '(%d, %d) -> %.8f' % (u, v, p)
... '(0, 1) -> 2.16404256'
... '(2, 3) -> 2.16404256'
```

### 4.25.4 preferential_attachment

**preferential_attachment**(G, ebunch=None)

Compute the preferential attachment score of all node pairs in ebunch.

Preferential attachment score of u and v is defined as

$$\frac{|\Gamma(u)||\Gamma(v)|}{|\Gamma(u)||\Gamma(v)|}$$

where $\Gamma(u)$ denotes the set of neighbors of u.

**Parameters**

G : graph

NetworkX undirected graph.

ebunch : iterable of node pairs, optional (default = None)

Preferential attachment score will be computed for each pair of nodes given in the iterable. The pairs must be given as 2-tuples (u, v) where u and v are nodes in the graph. If ebunch is None then all non-existent edges in the graph will be used. Default value: None.

**Returns**

piter : iterator

An iterator of 3-tuples in the form (u, v, p) where (u, v) is a pair of nodes and p is their preferential attachment score.

**References**

[R264]
Examples

```python
>>> import networkx as nx
>>> G = nx.complete_graph(5)
>>> preds = nx.preferential_attachment(G, [(0, 1), (2, 3)])
>>> for u, v, p in preds:
...    print('%d, %d -> %d' % (u, v, p))
...    print('(%d, %d) -> %d' % (u, v, p))
...    print('(0, 1) -> 16'
...    print('(2, 3) -> 16'
```

4.25.5 cn_soundarajan_hopcroft

`cn_soundarajan_hopcroft` (G, ebunch=None, community='community')

Count the number of common neighbors of all node pairs in ebunch using community information.

For two nodes `u` and `v`, this function computes the number of common neighbors and bonus one for each common neighbor belonging to the same community as `u` and `v`. Mathematically,

\[|\Gamma(u) \cap \Gamma(v)| + \sum_{w \in \Gamma(u) \cap \Gamma(v)} f(w)\]

where \( f(w) \) equals 1 if \( w \) belongs to the same community as `u` and `v` or 0 otherwise and \( \Gamma(u) \) denotes the set of neighbors of `u`.

**Parameters**

- **G**: graph
  A NetworkX undirected graph.
- **ebunch**: iterable of node pairs, optional (default = None)
  The score will be computed for each pair of nodes given in the iterable. The pairs must be given as 2-tuples `(u, v)` where `u` and `v` are nodes in the graph. If `ebunch` is None then all non-existent edges in the graph will be used. Default value: None.
- **community**: string, optional (default = 'community')
  Nodes attribute name containing the community information. `G[u][community]` identifies which community `u` belongs to. Each node belongs to at most one community. Default value: ‘community’.

**Returns**

- **piter**: iterator
  An iterator of 3-tuples in the form `(u, v, p)` where `(u, v)` is a pair of nodes and `p` is their score.

**References**

[R262]

**Examples**

```python
>>> import networkx as nx
>>> G = nx.path_graph(3)
>>> G.node[0]['community'] = 0
>>> G.node[1]['community'] = 0
```
>>> G.node[2]['community'] = 0
>>> preds = nx.cn_soundarajan_hopcroft(G, [(0, 2)])
>>> for u, v, p in preds:
...     '(%d, %d) -> %d' % (u, v, p)
...     '(0, 2) -> 2'

4.25.6 ra_index_soundarajan_hopcroft

ra_index_soundarajan_hopcroft (G, ebunch=None, community='community')

Compute the resource allocation index of all node pairs in ebunch using community information.

For two nodes $u$ and $v$, this function computes the resource allocation index considering only common neighbors belonging to the same community as $u$ and $v$. Mathematically,

$$
\sum_{w \in \Gamma(u) \cap \Gamma(v)} \frac{f(w)}{|\Gamma(w)|}
$$

where $f(w)$ equals 1 if $w$ belongs to the same community as $u$ and $v$ or 0 otherwise and $\Gamma(u)$ denotes the set of neighbors of $u$.

Parameters

- **G** : graph
  A NetworkX undirected graph.

- **ebunch** : iterable of node pairs, optional (default = None)
  The score will be computed for each pair of nodes given in the iterable. The pairs must be given as 2-tuples (u, v) where u and v are nodes in the graph. If ebunch is None then all non-existent edges in the graph will be used. Default value: None.

- **community** : string, optional (default = 'community')
  Nodes attribute name containing the community information. G[u][community] identifies which community u belongs to. Each node belongs to at most one community. Default value: 'community'.

Returns

**piter** : iterator
An iterator of 3-tuples in the form (u, v, p) where (u, v) is a pair of nodes and p is their score.

References

[R265]

Examples

>>> import networkx as nx
>>> G = nx.Graph()
>>> G.add_edges_from([(0, 1), (0, 2), (1, 3), (2, 3)])
>>> G.node[0]['community'] = 0
>>> G.node[1]['community'] = 0
>>> G.node[2]['community'] = 1
>>> G.node[3]['community'] = 0
>>> preds = nx.ra_index_soundarajan_hopcroft(G, [(0, 3)])
>>> for u, v, p in preds:
...    '(%d, %d) -> %.8f' % (u, v, p)
...    '(0, 3) -> 0.50000000'

4.25.7 within_inter_cluster

within_inter_cluster(G, ebunch=None, delta=0.001, community='community')

Compute the ratio of within- and inter-cluster common neighbors of all node pairs in ebunch.

For two nodes u and v, if a common neighbor w belongs to the same community as them, w is considered as within-cluster common neighbor of u and v. Otherwise, it is considered as inter-cluster common neighbor of u and v. The ratio between the size of the set of within- and inter-cluster common neighbors is defined as the WIC measure. [R267]

Parameters

G : graph
A NetworkX undirected graph.

ebunch : iterable of node pairs, optional (default = None)
The WIC measure will be computed for each pair of nodes given in the iterable. The pairs must be given as 2-tuples (u, v) where u and v are nodes in the graph. If ebunch is None then all non-existent edges in the graph will be used. Default value: None.

delta : float, optional (default = 0.001)
Value to prevent division by zero in case there is no inter-cluster common neighbor between two nodes. See [R267] for details. Default value: 0.001.

community : string, optional (default = ‘community’)
Nodes attribute name containing the community information. G[u][community] identifies which community u belongs to. Each node belongs to at most one community. Default value: ‘community’.

Returns

piter : iterator
An iterator of 3-tuples in the form (u, v, p) where (u, v) is a pair of nodes and p is their WIC measure.

References

[R267]

Examples

>>> import networkx as nx
>>> G = nx.Graph()
>>> G.add_edges_from([(0, 1), (0, 2), (0, 3), (1, 4), (2, 4), (3, 4)])
>>> G.node[0]['community'] = 0
>>> G.node[1]['community'] = 1
>>> G.node[2]['community'] = 0
>>> G.node[3]['community'] = 0
>>> G.node[4]['community'] = 0
>>> preds = nx.within_inter_cluster(G, [(0, 4)])
>>> for u, v, p in preds:
...

>>> preds = nx.within_inter_cluster(G, [(0, 4)], delta=0.5)

>>> for u, v, p

... in preds:

... '

...

'0, 4' \rightarrow 1.99800200'

'0, 4' \rightarrow 1.33333333'

4.26 Matching

| maximal_matching(G) | Find a maximal cardinality matching in the graph. |
| max_weight_matching(G[, maxcardinality]) | Compute a maximum-weighted matching of G. |

4.26.1 maximal_matching

maximal_matching(G)

Find a maximal cardinality matching in the graph.

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.

Parameters

G : NetworkX graph

Undirected graph

Returns

matching : set

A maximal matching of the graph.

Notes

The algorithm greedily selects a maximal matching M of the graph G (i.e. no superset of M exists). It runs in \(O(|E|)\) time.

4.26.2 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.

This method 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 [R268].

References

[R268]

4.27 Maximal independent set

Algorithm to find a maximal (not maximum) independent set.

\[\text{maximal_independent_set}(G, \text{nodes} = \text{None})\]

Return a random maximal independent set guaranteed to contain a given set of nodes.

4.27.1 maximal_independent_set

Return a random maximal independent set guaranteed to contain a given set of nodes.

An independent set is a set of nodes such that the subgraph of G induced by these nodes contains no edges. A maximal independent set is an independent set such that it is not possible to add a new node and still get an independent set.

Parameters

- **G**: NetworkX graph
- **nodes**: list or iterable
  
  Nodes that must be part of the independent set. This set of nodes must be independent.

Returns

- **indep_nodes**: list
  
  List of nodes that are part of a maximal independent set.

Raises

- **NetworkXUnfeasible**
  
  If the nodes in the provided list are not part of the graph or do not form an independent set, an exception is raised.

Notes

This algorithm does not solve the maximum independent set problem.
Examples

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

4.28 Minimum Spanning Tree

Computes minimum spanning tree of a weighted graph.

```python
minimum_spanning_tree(G[, weight])
```

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

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

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

Parameters

- **G**: NetworkX Graph
- **weight**: string

Returns

- **G**: NetworkX Graph

A minimum spanning tree or forest.

Notes

Uses Kruskal's algorithm.

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

Examples

```python
>>> G=nx.cycle_graph(4)
>>> G.add_edge(0,3,weight=2)  # assign weight 2 to edge 0-3
>>> T=nx.minimum_spanning_tree(G)
>>> print(sorted(T.edges(data=True)))
[(0, 1, {}), (1, 2, {}), (2, 3, {})]
```
4.28.2 minimum_spanning_edges

minimum_spanning_edges \((G, \text{weight}='\text{weight}', \text{data}=\text{True})\)
Generate edges in a minimum spanning forest of an undirected weighted graph.

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

Parameters

G : NetworkX Graph
weight : string
   Edge data key to use for weight (default ‘weight’).

Returns

edges : iterator
   A generator that produces edges in the minimum spanning tree. The edges are three-tuples (u,v,w) where w is the weight.

Notes

Uses Kruskal’s algorithm.

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

Modified code from David Eppstein, April 2006 http://www.ics.uci.edu/~eppstein/PADS/

Examples

```python
>>> G = nx.cycle_graph(4)
>>> G.add_edge(0, 3, weight=2)  # assign weight 2 to edge 0-3
>>> mst = nx.minimum_spanning_edges(G, data=False)  # a generator of MST edges
>>> edgelist = list(mst)  # make a list of the edges
>>> print(sorted(edgelist))
[(0, 1), (1, 2), (2, 3)]
```

4.29 Operators

Unary operations on graphs

<table>
<thead>
<tr>
<th>operator</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>complement</td>
<td>Return the graph complement of G.</td>
</tr>
<tr>
<td>reverse</td>
<td>Return the reverse directed graph of G.</td>
</tr>
</tbody>
</table>

4.29.1 complement

complement \((G, \text{name}=None)\)
Return the graph complement of G.

Parameters

G : graph
   A NetworkX graph
**name** : string

Specify name for new graph

**Returns**  GC : A 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.

### 4.29.2 reverse

**reverse** *(G, copy=True)*

Return the reverse directed graph of G.

**Parameters**  G : directed graph

A NetworkX directed graph

**copy** : bool

If True, then a new graph is returned. If False, then the graph is reversed in place.

**Returns**  H : directed graph

The reversed G.

Operations on graphs including union, intersection, difference.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>compose(G, H[, name])</td>
<td>Return a new graph of G composed with H.</td>
</tr>
<tr>
<td>union(G, H[, rename, name])</td>
<td>Return the union of graphs G and H.</td>
</tr>
<tr>
<td>disjoint_union(G, H)</td>
<td>Return the disjoint union of graphs G and H.</td>
</tr>
<tr>
<td>intersection(G, H)</td>
<td>Return a new graph that contains only the edges that exist in G and H.</td>
</tr>
<tr>
<td>difference(G, H)</td>
<td>Return a new graph that contains the edges that exist in G but not in H.</td>
</tr>
<tr>
<td>symmetric_difference(G, H)</td>
<td>Return new graph with edges that exist in either G or H but not both.</td>
</tr>
</tbody>
</table>

### 4.29.3 compose

**compose** *(G, H, 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

**name** : string

Specify name for new graph

**Returns**  C: A new graph with the same type as G
Notes

It is recommended that G and H be either both directed or both undirected. Attributes from H take precedence over attributes from G.

4.29.4 union

union \((G, H, rename=(None, None), name=\text{None})\)

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
- **rename**: bool, default=(None, None)
  - Node names of G and H can be changed by specifying the tuple rename=('G-','H-') (for example). Node “u” in G is then renamed “G-u” and “v” in H is renamed “H-v”.
- **name**: string
  - Specify the name for the union graph

Returns

- **U**: A union graph with the same type as G.

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 H is used.

4.29.5 disjoint_union

disjoint_union \((G, H)\)

Return the disjoint union of graphs G and H.

This algorithm forces distinct integer node labels.

Parameters

- **G,H**: graph
  - A NetworkX graph

Returns

- **U**: A union graph with the same type as G.
### 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.

The nodes of G are relabeled 0 to len(G)-1, and the nodes of H are relabeled len(G) to len(G)+len(H)-1.

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 H is used.

#### 4.29.6 intersection

**intersection** (*G, H*)

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.

**Returns**  
GH : A new graph 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:

```python
>>> 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)
```

#### 4.29.7 difference

**difference** (*G, H*)

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.

**Returns**  
D : A new graph 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 the attributes (including edge data) from G use remove_nodes_from() as follows:

```python
>>> 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)
```
4.29.8 symmetric_difference

**symmetric_difference**\((G, H)\)

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.

**Returns**  
D : A new graph with the same type as G.

**Notes**

Attributes from the graph, nodes, and edges are not copied to the new graph.

Operations on many graphs:

- **compose_all**\((graphs, name=None)\)  
  Return the composition of all graphs.

- **union_all**\((graphs, rename=(None, ), name=None)\)  
  Return the union of all graphs.

- **disjoint_union_all**\((graphs)\)  
  Return the disjoint union of all graphs.

- **intersection_all**\((graphs)\)  
  Return a new graph that contains only the edges that exist in all graphs.

4.29.9 compose_all

**compose_all**\((graphs, name=None)\)

Return the composition of all graphs.

Composition is the simple union of the node sets and edge sets. The node sets of the supplied graphs need not be disjoint.

**Parameters**  
\(graphs\) : list  
List of NetworkX graphs

\(name\) : string  
Specify name for new graph

**Returns**  
C : A graph with the same type as the first graph in list

**Notes**

It is recommended that the supplied graphs be either all directed or all undirected.

Graph, edge, and node attributes are propagated to the union graph. If a graph attribute is present in multiple graphs, then the value from the last graph in the list with that attribute is used.

4.29.10 union_all

**union_all**\((graphs, rename=(None, ), name=None)\)

Return the union of all graphs.

The graphs must be disjoint, otherwise an exception is raised.

**Parameters**  
\(graphs\) : list of graphs
List of NetworkX graphs

rename : bool, default=(None, None)

Node names of G and H can be changed by specifying the tuple rename=('G-','H-') (for example). Node “u” in G is then renamed “G-u” and “v” in H is renamed “H-v”.

name : string

Specify the name for the union graph

Returns U : a graph with the same type as the first graph in list

See Also:
union, disjoint_union_all

Notes

To force a disjoint union with node relabeling, use disjoint_union_all(G,H) or convert_node_labels_to_integers(). Graph, edge, and node attributes are propagated to the union graph. If a graph attribute is present in multiple graphs, then the value from the last graph in the list with that attribute is used.

4.29.11 disjoint_union_all

disjoint_union_all (graphs)
Return the disjoint union of all graphs.
This operation forces distinct integer node labels starting with 0 for the first graph in the list and numbering consecutively.

Parameters graphs : list
List of NetworkX graphs

Returns U : A graph with the same type as the first graph in list

Notes

It is recommended that the graphs be either all directed or all undirected.
Graph, edge, and node attributes are propagated to the union graph. If a graph attribute is present in multiple graphs, then the value from the last graph in the list with that attribute is used.

4.29.12 intersection_all

intersection_all (graphs)
Return a new graph that contains only the edges that exist in all graphs.
All supplied graphs must have the same node set.

Parameters graphs_list : list
List of NetworkX graphs

Returns R : A new graph with the same type as the first graph in list
NetworkX Reference, Release 1.9

Notes

Attributes from the graph, nodes, and edges are not copied to the new graph.

Graph products.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cartesian_product(G, H)</td>
<td>Return the Cartesian product of G and H.</td>
</tr>
<tr>
<td>lexicographic_product(G, H)</td>
<td>Return the lexicographic product of G and H.</td>
</tr>
<tr>
<td>strong_product(G, H)</td>
<td>Return the strong product of G and H.</td>
</tr>
<tr>
<td>tensor_product(G, H)</td>
<td>Return the tensor product of G and H.</td>
</tr>
</tbody>
</table>

4.29.13 cartesian_product

cartesian_product \((G, H)\)

Return the Cartesian product of \(G\) and \(H\).

The tensor product \(P\) of the graphs \(G\) and \(H\) has a node set that is the Cartesian product of the node sets, \(\text{SV}(P) = \text{V}(G) \times \text{V}(H)\). \(P\) has an edge \(((u,v),(x,y))\) if and only if \((u,v)\) is an edge in \(G\) and \(x = y\) or and \((x,y)\) is an edge in \(H\) and \(u = v\).

Parameters  
\(G, H: \text{graphs}\)

Networkx graphs.

Returns  
\(P: \text{NetworkX graph}\)

The Cartesian product of \(G\) and \(H\). \(P\) will be a multi-graph if either \(G\) or \(H\) is a multi-graph. Will be a directed if \(G\) and \(H\) are directed, and undirected if \(G\) and \(H\) are undirected.

Raises  
NetworkXError

If \(G\) and \(H\) are not both directed or both undirected.

Notes

Node attributes in \(P\) are two-tuple of the \(G\) and \(H\) node attributes. Missing attributes are assigned None.

For example

```
>>> G = nx.Graph()
>>> H = nx.Graph()
>>> G.add_node(0,a1=True)
>>> H.add_node('a',a2='Spam')
>>> P = nx.cartesian_product(G,H)
>>> P.nodes()
[(0, 'a')]
```

Edge attributes and edge keys (for multigraphs) are also copied to the new product graph.

4.29.14 lexicographic_product

lexicographic_product \((G, H)\)

Return the lexicographical product of \(G\) and \(H\).

The lexicographical product \(P\) of the graphs \(G\) and \(H\) has a node set that is the Cartesian product of the node sets, \(\text{SV}(P) = \text{V}(G) \times \text{V}(H)\). \(P\) has an edge \(((u,v),(x,y))\) if and only if \((u,v)\) is an edge in \(G\) or \(u = v\) and \((x,y)\) is an edge in \(H\).

Parameters  
\(G, H: \text{graphs}\)

Networkx graphs.

Returns  
\(P: \text{NetworkX graph}\)
The Cartesian product of G and H. P will be a multi-graph if either G or H is a multi-
graph. Will be a directed if G and H are directed, and undirected if G and H are undi-
rected.

Raises NetworkXError
If G and H are not both directed or both undirected.

Notes
Node attributes in P are two-tuple of the G and H node attributes. Missing attributes are assigned None.
For example >>> G = nx.Graph() >>> H = nx.Graph() >>> G.add_node(0,a1=True) >>>
H.add_node(‘a’,a2=’Spam’) >>> P = nx.lexicographic_product(G,H) >>> P.nodes() 
[(0, ‘a’)]

Edge attributes and edge keys (for multigraphs) are also copied to the new product graph

4.29.15 strong_product

strong_product (G, H)
Return the strong product of G and H.
The strong product P of the graphs G and H has a node set that is the Cartesian product of the node sets,
$V(P)=V(G) \times V(H)$. P has an edge ((u,v),(x,y)) if and only if u==v and (x,y) is an edge in H, or x==y and
(u,v) is an edge in G, or (u,v) is an edge in G and (x,y) is an edge in H.

Parameters G, H: graphs
Networkx graphs.

Returns P: NetworkX graph
The Cartesian product of G and H. P will be a multi-graph if either G or H is a multi-
graph. Will be a directed if G and H are directed, and undirected if G and H are undi-
crected.

Raises NetworkXError
If G and H are not both directed or both undirected.

Notes
Node attributes in P are two-tuple of the G and H node attributes. Missing attributes are assigned None.
For example >>> G = nx.Graph() >>> H = nx.Graph() >>> G.add_node(0,a1=True) >>>
H.add_node(‘a’,a2=’Spam’) >>> P = nx.strong_product(G,H) >>> P.nodes() 
[(0, ‘a’)]

Edge attributes and edge keys (for multigraphs) are also copied to the new product graph

4.29.16 tensor_product

tensor_product (G, H)
Return the tensor product of G and H.
The tensor product P of the graphs G and H has a node set that is the Cartesian product of the node sets,
$V(P)=V(G) \times V(H)$. P has an edge ((u,v),(x,y)) if and only if (u,v) is an edge in G and (x,y) is an edge in H.
Sometimes referred to as the categorical product.

**Parameters**  
G, H: graphs  
Networkx graphs.

**Returns**  
P: NetworkX graph  
The tensor product of G and H. P will be a multi-graph if either G or H is a multi-graph. Will be a directed if G and H are directed, and undirected if G and H are undirected.

**Raises**  
NetworkXError  
If G and H are not both directed or both undirected.

**Notes**  
Node attributes in P are two-tuple of the G and H node attributes. Missing attributes are assigned None.

For example  
```python  
>>> G = nx.Graph()  
>>> H = nx.Graph()  
>>> G.add_node(0,a1=True)  
>>> H.add_node('a',a2='Spam')  
>>> P = nx.tensor_product(G,H)  
>>> P.nodes()  
[(0, 'a')]  
```

Edge attributes and edge keys (for multigraphs) are also copied to the new product graph.

---

### 4.30 Rich Club

**rich_club_coefficient**(G[, normalized, Q])  
Return the rich-club coefficient of the graph G.

The rich-club coefficient is the ratio, for every degree k, of the number of actual to the number of potential edges for nodes with degree greater than k:

\[ \phi(k) = \frac{2E_k}{N_k(N_k - 1)} \]

where \(N_k\) is the number of nodes with degree larger than k, and \(E_k\) be the number of edges among those nodes.

**Parameters**  
G : NetworkX graph  
normalized : bool (optional)  
Normalize using randomized network (see [R269])  
Q : float (optional, default=100)  
If normalized=True build a random network by performing Q*M double-edge swaps, where M is the number of edges in G, to use as a null-model for normalization.

**Returns**  
rc : dictionary  
A dictionary, keyed by degree, with rich club coefficient values.
Notes

The rich club definition and algorithm are found in [R269]. This algorithm ignores any edge weights and is not defined for directed graphs or graphs with parallel edges or self loops.

Estimates for appropriate values of Q are found in [R270].

References

[R269], [R270]

Examples

```python
>>> G = nx.Graph([(0,1),(0,2),(1,2),(1,3),(1,4),(4,5)])
>>> rc = nx.rich_club_coefficient(G, normalized=False)
>>> rc[0]
0.4
```

4.31 Shortest Paths

Compute the shortest paths and path lengths between nodes in the graph.

These algorithms work with undirected and directed graphs.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>shortest_path(G[, source, target, weight])</code></td>
<td>Compute shortest paths in the graph.</td>
</tr>
<tr>
<td><code>all_shortest_paths(G, source, target[, weight])</code></td>
<td>Compute all shortest paths in the graph.</td>
</tr>
<tr>
<td><code>shortest_path_length(G[, source, target, weight])</code></td>
<td>Compute shortest path lengths in the graph.</td>
</tr>
<tr>
<td><code>average_shortest_path_length(G[, weight])</code></td>
<td>Return the average shortest path length.</td>
</tr>
<tr>
<td><code>has_path(G, source, target)</code></td>
<td>Return True if G has a path from source to target, False otherwise.</td>
</tr>
</tbody>
</table>

4.31.1 shortest_path

`shortest_path(G, source=None, target=None, weight=None)`  
Compute shortest paths in the graph.

Parameters  

- **G** : NetworkX graph  
  - **source** : node, optional  
    Starting node for path. If not specified, compute shortest paths using all nodes as source nodes.  
  - **target** : node, optional  
    Ending node for path. If not specified, compute shortest paths using all nodes as target nodes.  
  - **weight** : None or string, optional (default = None)  
    If None, every edge has weight/distance/cost 1. If a string, use this edge attribute as the edge weight. Any edge attribute not present defaults to 1.

Returns  

- path: list or dictionary
All returned paths include both the source and target in the path.

If the source and target are both specified, return a single list of nodes in a shortest path from the source to the target.

If only the source is specified, return a dictionary keyed by targets with a list of nodes in a shortest path from the source to one of the targets.

If only the target is specified, return a dictionary keyed by sources with a list of nodes in a shortest path from one of the sources to the target.

If neither the source nor target are specified return a dictionary of dictionaries with path[source][target]=list of nodes in path.

See Also:

all_pairs_shortest_path, all_pairs_dijkstra_path, single_source_shortest_path, single_source_dijkstra_path

Notes

There may be more than one shortest path between a source and target. This returns only one of them.

Examples

```python
>>> 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,target=4)  # source not specified
>>> p[0]
[0, 1, 2, 3, 4]
>>> p=nx.shortest_path(G)  # source,target not specified
>>> p[0][4]
[0, 1, 2, 3, 4]
```

4.31.2 all_shortest_paths

all_shortest_paths (G, source, target, weight=None)

Compute all shortest paths in the graph.

Parameters  

G : NetworkX graph

source : node

Starting node for path.

target : node

Ending node for path.

weight : None or string, optional (default = None)

If None, every edge has weight/distance/cost 1. If a string, use this edge attribute as the edge weight. Any edge attribute not present defaults to 1.

Returns  

paths: generator of lists
A generator of all paths between source and target.

**See Also:**

`shortest_path`, `single_source_shortest_path`, `all_pairs_shortest_path`

**Notes**

There may be many shortest paths between the source and target.

**Examples**

```python
>>> G = nx.Graph()
>>> G.add_path([0,1,2])
>>> G.add_path([0,10,2])
>>> print([p for p in nx.all_shortest_paths(G, source=0, target=2)])
[[0, 1, 2], [0, 10, 2]]
```

### 4.31.3 shortest_path_length

**shortest_path_length** *(G, source=None, target=None, weight=None)*

Compute shortest path lengths in the graph.

**Parameters**

- **G** : NetworkX graph
  - source : node, optional
    - Starting node for path. If not specified, compute shortest path lengths using all nodes as source nodes.
  - target : node, optional
    - Ending node for path. If not specified, compute shortest path lengths using all nodes as target nodes.
  - weight : None or string, optional (default = None)
    - If None, every edge has weight/distance/cost 1. If a string, use this edge attribute as the edge weight. Any edge attribute not present defaults to 1.

**Returns**

- **length** : int or dictionary
  - If the source and target are both specified, return the length of the shortest path from the source to the target.
  - If only the source is specified, return a dictionary keyed by targets whose values are the lengths of the shortest path from the source to one of the targets.
  - If only the target is specified, return a dictionary keyed by sources whose values are the lengths of the shortest path from one of the sources to the target.
  - If neither the source nor target are specified return a dictionary of dictionaries with path[source][target]=L, where L is the length of the shortest path from source to target.

**Raises**

- NetworkXNoPath
  - If no path exists between source and target.
See Also:

all_pairs_shortest_path_length, all_pairs_dijkstra_path_length,
single_source_shortest_path_length, single_source_dijkstra_path_length

Notes

The length of the path is always 1 less than the number of nodes involved in the path since the length measures
the number of edges followed.

For digraphs this returns the shortest directed path length. To find path lengths in the reverse direction use
G.reverse(copy=False) first to flip the edge orientation.

Examples

```python
>>> 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,target=4)  # source not specified
>>> p[0]
4
>>> p=nx.shortest_path_length(G)  # source, target not specified
>>> p[0][4]
4
```

4.31.4 average_shortest_path_length

average_shortest_path_length(G, weight=None)

Return the average shortest path length.

The average shortest path length is

\[
\alpha = \frac{\sum_{s,t\in V} d(s,t)}{n(n-1)}
\]

where \( V \) is the set of nodes in \( G \), \( d(s,t) \) is the shortest path from \( s \) to \( t \), and \( n \) is the number of nodes in \( G \).

Parameters

- **G**: NetworkX graph
- **weight**: None or string, optional (default = None)

If None, every edge has weight/distance/cost 1. If a string, use this edge attribute as the
edge weight. Any edge attribute not present defaults to 1.

Raises

NetworkXError:

if the graph is not connected.

Examples

```python
>>> G=nx.path_graph(5)
>>> print(nx.average_shortest_path_length(G))
2.0
```
For disconnected graphs you can compute the average shortest path length for each component:
```python
>>> G = nx.Graph([(1, 2), (3, 4)])
>>> for g in nx.connected_component_subgraphs(G):
...    print(nx.average_shortest_path_length(g))
1.0
1.0
```

### 4.31.5 has_path

**has_path** *(G, source, target)*

Return True if G has a path from source to target, False otherwise.

**Parameters**

- **G** : NetworkX graph
- **source** : node
  - Starting node for path
- **target** : node
  - Ending node for path

### 4.31.6 Advanced Interface

Shortest path algorithms for unweighted graphs.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>single_source_shortest_path</code></td>
<td>Compute shortest path between source and all other nodes reachable from source.</td>
</tr>
<tr>
<td><code>single_source_shortest_path_length</code></td>
<td>Compute the shortest path lengths from source to all reachable nodes.</td>
</tr>
<tr>
<td><code>all_pairs_shortest_path</code></td>
<td>Compute shortest paths between all nodes.</td>
</tr>
<tr>
<td><code>all_pairs_shortest_path_length</code></td>
<td>Compute the shortest path lengths between all nodes in G.</td>
</tr>
<tr>
<td><code>predecessor</code></td>
<td>Returns dictionary of predecessors for the path from source to all nodes.</td>
</tr>
</tbody>
</table>

**single_source_shortest_path** *(G, source[, cutoff])*

Compute shortest path between source and all other nodes reachable from source.

**Parameters**

- **G** : NetworkX graph
- **source** : node label
  - Starting node for path
- **cutoff** : integer, optional
  - Depth to stop the search. Only paths of length <= cutoff are returned.

**Returns**

- **lengths** : dictionary
  - Dictionary, keyed by target, of shortest paths.

**See Also:**

`shortest_path`

**Notes**

The shortest path is not necessarily unique. So there can be multiple paths between the source and each target node, all of which have the same ‘shortest’ length. For each target node, this function returns only one of those paths.
Examples

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

**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:**

- `single_source_shortest_path_length`

Examples

```python
>>> 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}
```

**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

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

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

```python
>>> 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}
```

predecessor

`predecessor(G, source=None, 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**

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

Shortest path algorithms for weighed graphs.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dijkstra_path(G, source, target[, weight])</code></td>
<td>Returns the shortest path from source to target in a weighted graph G.</td>
</tr>
<tr>
<td><code>dijkstra_path_length(G, source, target[, weight])</code></td>
<td>Returns the shortest path length from source to target in a weighted graph.</td>
</tr>
<tr>
<td><code>single_source_dijkstra_path(G, source[, ...])</code></td>
<td>Compute shortest path between source and all other reachable nodes.</td>
</tr>
<tr>
<td><code>single_source_dijkstra_path_length(G, source)</code></td>
<td>Compute the shortest path length between source and all other reachable nodes.</td>
</tr>
<tr>
<td><code>all_pairs_dijkstra_path(G[, cutoff, weight])</code></td>
<td>Compute shortest paths between all nodes in a weighted graph.</td>
</tr>
<tr>
<td><code>all_pairs_dijkstra_path_length(G[, cutoff, ...])</code></td>
<td>Compute shortest path lengths between all nodes in a weighted graph.</td>
</tr>
<tr>
<td><code>single_source_dijkstra(G, source[, target, ...])</code></td>
<td>Compute shortest paths and lengths in a weighted graph G.</td>
</tr>
<tr>
<td><code>bidirectional_dijkstra(G, source, target[, ...])</code></td>
<td>Dijkstra's algorithm for shortest paths using bidirectional search.</td>
</tr>
<tr>
<td><code>bellman_ford(G, source[, weight])</code></td>
<td>Compute shortest path length and predecessors on shortest paths in weighted graph.</td>
</tr>
<tr>
<td><code>negative_edge_cycle(G[, weight])</code></td>
<td>Return True if there exists a negative edge cycle anywhere in G.</td>
</tr>
</tbody>
</table>

### `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 (default='weight')
  Edge data key corresponding to the edge weight

**Returns**

- **path** : list
  List of nodes in a shortest path.

**Raises**

- `NetworkXNoPath`
  If no path exists between source and target.

**See Also:**

- `bidirectional_dijkstra`

**Notes**

Edge weight attributes must be numerical. Distances are calculated as sums of weighted edges traversed.
Examples

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

dijkstra_path_length

`dijkstra_path_length(G, source, target, weight='weight')`

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

**Parameters**

- `G`: NetworkX graph
- `source`: node label
  - starting node for path
- `target`: node label
  - ending node for path
- `weight`: string, optional (default=`'weight'`)
  - Edge data key corresponding to the edge weight

**Returns**

- `length`: number
  - Shortest path length.

**Raises**

- `NetworkXNoPath`
  - If no path exists between source and target.

**See Also**

- `bidirectional_dijkstra`

**Notes**

Edge weight attributes must be numerical. Distances are calculated as sums of weighted edges traversed.

**Examples**

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

single_source_dijkstra_path

`single_source_dijkstra_path(G, source, cutoff=None, weight='weight')`

Compute shortest path between source and all other reachable nodes for a weighted graph.

**Parameters**

- `G`: NetworkX graph
- `source`: node
  - Starting node for path.
- `weight`: string, optional (default=`'weight'`)
Edge data key corresponding to the edge weight

cutoff : integer or float, optional
    Depth to stop the search. Only paths of length <= cutoff are returned.

Returns paths : dictionary
    Dictionary of shortest path lengths keyed by target.

See Also:

single_source_dijkstra

Notes

Edge weight attributes must be numerical. Distances are calculated as sums of weighted edges traversed.

Examples

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

single_source_dijkstra_path_length

single_source_dijkstra_path_length(G, source, cutoff=None, weight='weight')
    Compute the shortest path length between source and all other reachable nodes for a weighted graph.

Parameters  G : NetworkX graph
    source : node label
        Starting node for path

weight : string, optional (default='weight')
    Edge data key corresponding to the edge weight.

cutoff : integer or float, optional
    Depth to stop the search. Only paths of length <= cutoff are returned.

Returns length : dictionary
    Dictionary of shortest lengths keyed by target.

See Also:

single_source_dijkstra

Notes

Edge weight attributes must be numerical. Distances are calculated as sums of weighted edges traversed.
Examples

```python
>>> G=nx.path_graph(5)
>>> length=nx.single_source_dijkstra_path_length(G,0)
4
>>> print(length)
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4}
```

all_pairs_dijkstra_path

all_pairs_dijkstra_path(G, cutoff=None, weight='weight')

Compute shortest paths between all nodes in a weighted graph.

Parameters

- **G**: NetworkX graph
- **weight**: string, optional (default='weight')
  - Edge data key corresponding to the edge weight
- **cutoff**: integer or float, optional
  - Depth to stop the search. Only paths of length <= cutoff are returned.

Returns

- **distance**: dictionary
  - Dictionary, keyed by source and target, of shortest paths.

See Also:

floyd_warshall

Notes

Edge weight attributes must be numerical. Distances are calculated as sums of weighted edges traversed.

Examples

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

all_pairs_dijkstra_path_length

all_pairs_dijkstra_path_length(G, cutoff=None, weight='weight')

Compute shortest path lengths between all nodes in a weighted graph.

Parameters

- **G**: NetworkX graph
- **weight**: string, optional (default='weight')
  - Edge data key corresponding to the edge weight
- **cutoff**: integer or float, optional
  - Depth to stop the search. Only paths of length <= cutoff are returned.
Returns  **distance** : dictionary

Dictionary, keyed by source and target, of shortest path lengths.

Notes

Edge weight attributes must be numerical. Distances are calculated as sums of weighted edges traversed. The dictionary returned only has keys for reachable node pairs.

Examples

```python
>>> 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}
```

**single_source_dijkstra**

**single_source_dijkstra** \((G, \text{source}, \text{target}=\text{None, cutoff}=\text{None, weight}='\text{weight}')\)

Compute shortest paths and lengths in a weighted graph \(G\).

Uses Dijkstra’s algorithm for shortest paths.

Parameters  

\(G\) : NetworkX graph

source : node label
Starting node for path

target : node label, optional
Ending node for path

cutoff : integer or float, optional
Depth to stop the search. Only paths of length \(\leq\) cutoff are returned.

Returns  **distance, path** : dictionaries

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

See Also:  

**single_source_dijkstra_path, single_source_dijkstra_path_length**

Notes

Edge weight attributes must be numerical. Distances are calculated as sums of weighted edges traversed.

Based on the Python cookbook recipe (119466) at http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/119466

This algorithm is not guaranteed to work if edge weights are negative or are floating point numbers (overflows and roundoff errors can cause problems).
Examples

```python
>>> 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]
```

**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 (default=`'weight'`)
  - Edge data key corresponding to the edge weight

**Returns**

- `length` : number
  - Shortest path length.
- Returns a tuple of two dictionaries keyed by node.
  - The first dictionary stores distance from the source.
  - The second stores the path from the source to that node.

**Raises**

- `NetworkXNoPath`
  - If no path exists between source and target.

**See Also:**

- `shortest_path`, `shortest_path_length`

**Notes**

Edge weight attributes must be numerical. Distances are calculated as sums of weighted edges traversed.

In practice bidirectional Dijkstra is much more than twice as fast as ordinary Dijkstra.

Ordinary Dijkstra expands nodes in a sphere-like manner from the source. The radius of this sphere will eventually be the length of the shortest path. Bidirectional Dijkstra will expand nodes from both the source and the target, making two spheres of half this radius. Volume of the first sphere is \( \pi r^2 r \) 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

```python
>>> G = nx.path_graph(5)
>>> length, path = nx.bidirectional_dijkstra(G, 0, 4)
>>> print(length)
4
>>> print(path)
[0, 1, 2, 3, 4]
```

dijkstra_predecessor_and_distance

dijkstra_predecessor_and_distance(G, source, cutoff=None, weight='weight')

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

**Parameters**

- `G` : NetworkX graph
- `source` : node label
- `cutoff` : integer or float, optional
  - Depth to stop the search. Only paths of length <= cutoff are returned.
- `weight` : string, optional (default='weight')

**Returns**

- `pred` : dictionary
  - Returns a dictionary representing a list of predecessors of a node.
- `distance` : dictionary
  - Returns a dictionary representing the distance to each node.

**Notes**

Edge weight attributes must be numerical. Distances are calculated as sums of weighted edges traversed.

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

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 m is the number of edges. It is slower than Dijkstra but can handle negative edge weights.

**Parameters**

- `G` : NetworkX graph
- `source` : node label
- `weight` : string, optional (default='weight')

The algorithm works for all types of graphs, including directed graphs and multigraphs.
Returns \( \text{pred, dist} \): dictionaries

Returns two dictionaries keyed by node to predecessor in the path and to the distance from the source respectively.

Raises \texttt{NetworkXUnbounded}

If the (di)graph contains a negative cost (di)cycle, the algorithm raises an exception to indicate the presence of the negative cost (di)cycle. Note: any negative weight edge in an undirected graph is a negative cost cycle.

Notes

Edge weight attributes must be numerical. Distances are calculated as sums of weighted edges traversed.

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

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

Examples

```python
>>> import networkx as nx
>>> G = nx.path_graph(5, create_using = nx.DiGraph())
>>> pred, dist = nx.bellman_ford(G, 0)
>>> sorted(pred.items())
[(0, None), (1, 0), (2, 1), (3, 2), (4, 3)]
>>> sorted(dist.items())
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4)]
```

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

\texttt{negative_edge_cycle}

\texttt{negative_edge_cycle}(G, weight='weight')

Return True if there exists a negative edge cycle anywhere in G.

Parameters G : NetworkX graph

weight: string, optional (default='weight')

Edge data key corresponding to the edge weight

Returns negative_cycle: bool

True if a negative edge cycle exists, otherwise False.

Notes

Edge weight attributes must be numerical. Distances are calculated as sums of weighted edges traversed.

This algorithm uses \texttt{bellman_ford()} but finds negative cycles on any component by first adding a new node connected to every node, and starting \texttt{bellman_ford} on that node. It then removes that extra node.
Examples

```python
>>> import networkx as nx
>>> G = nx.cycle_graph(5, create_using = nx.DiGraph())
>>> print(nx.negative_edge_cycle(G))
False
>>> G[1][2]['weight'] = -7
>>> print(nx.negative_edge_cycle(G))
True
```

4.31.7 Dense Graphs

Floyd-Warshall algorithm for shortest paths.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>floyd_warshall(G[, weight])</code></td>
<td>Find all-pairs shortest path lengths using Floyd’s algorithm.</td>
</tr>
<tr>
<td><code>floyd_warshall_predecessor_and_distance(G[, ...])</code></td>
<td>Find all-pairs shortest path lengths using Floyd’s algorithm.</td>
</tr>
<tr>
<td><code>floyd_warshall_numpy(G[, nodelist, weight])</code></td>
<td>Find all-pairs shortest path lengths using Floyd’s algorithm.</td>
</tr>
</tbody>
</table>

### Floyd_warshall

`floyd_warshall(G, weight='weight')`

Find all-pairs shortest path lengths using Floyd’s algorithm.

**Parameters**

- `G` : NetworkX graph
- `weight` : string, optional (default= ‘weight’)  
  Edge data key corresponding to the edge weight.

**Returns**

- `distance` : dict  
  A dictionary, keyed by source and target, of shortest paths distances between nodes.

**See Also:**

- `floyd_warshall_predecessor_and_distance`
- `floyd_warshall_numpy`
- `all_pairs_shortest_path`
- `all_pairs_shortest_path_length`

**Notes**

Floyd’s algorithm is appropriate for finding shortest paths in dense graphs or graphs with negative weights when Dijkstra’s algorithm fails. This algorithm can still fail if there are negative cycles. It has running time O(n^3) with running space of O(n^2).

### Floyd_warshall_predecessor_and_distance

`floyd_warshall_predecessor_and_distance(G, weight='weight')`

Find all-pairs shortest path lengths using Floyd’s algorithm.

**Parameters**

- `G` : NetworkX graph
- `weight` : string, optional (default= ‘weight’)  
  Edge data key corresponding to the edge weight.

**Returns**

- `predecessor,distance` : dictionaries

4.31. Shortest Paths 323
Dictionaries, keyed by source and target, of predecessors and distances in the shortest path.

See Also:

floyd_warshall, floyd_warshall_numpy, all_pairs_shortest_path, all_pairs_shortest_path_length

Notes

Floyd’s algorithm is appropriate for finding shortest paths in dense graphs or graphs with negative weights when Dijkstra’s algorithm fails. This algorithm can still fail if there are negative cycles. It has running time $O(n^3)$ with running space of $O(n^2)$.

floyd_warshall_numpy

`floyd_warshall_numpy(G, nodelist=None, weight='weight')`

Find all-pairs shortest path lengths using Floyd’s algorithm.

Parameters

$G$ : NetworkX graph

nodelist : list, optional

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

weight : string, optional (default= ‘weight’)

Edge data key corresponding to the edge weight.

Returns

distance : NumPy matrix

A matrix of shortest path distances between nodes. If there is no path between to nodes the corresponding matrix entry will be Inf.

Notes

Floyd’s algorithm is appropriate for finding shortest paths in dense graphs or graphs with negative weights when Dijkstra’s algorithm fails. This algorithm can still fail if there are negative cycles. It has running time $O(n^3)$ with running space of $O(n^2)$.

4.31.8 A* Algorithm

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

`astar_path(G, source, target[, heuristic, ...])` Return a list of nodes in a shortest path between source and target

`astar_path_length(G, source, target[, ...])` Return the length of the shortest path between source and target using

astar_path

`astar_path(G, source, target, heuristic=None, weight='weight')`

Return a list of nodes in a shortest path between source and target using the A* (“A-star”) algorithm.

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

- **G**: NetworkX graph
- **source**: node
  - Starting node for path
- **target**: node
  - Ending node for path
- **heuristic**: function
  - A function to evaluate the estimate of the distance from the a node to the target. The function takes two nodes arguments and must return a number.
- **weight**: string, optional (default='weight')
  - Edge data key corresponding to the edge weight.

Raises  

- **NetworkXNoPath**: If no path exists between source and target.

See Also:

- `shortest_path`, `dijkstra_path`

Examples

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

### astar_path_length

`astar_path_length(G, source, target, heuristic=None, weight='weight')`

Return the length of the shortest path between source and target using the A* ("A-star") algorithm.

Parameters  

- **G**: NetworkX graph
- **source**: node
  - Starting node for path
- **target**: node
  - Ending node for path
- **heuristic**: function
  - A function to evaluate the estimate of the distance from the a node to the target. The function takes two nodes arguments and must return a number.

Raises  

- **NetworkXNoPath**: If no path exists between source and target.
4.32 Simple Paths

**all_simple_paths** (G, source, target[, cutoff])  Generate all simple paths in the graph G from source to target.

A simple path is a path with no repeated nodes.

Parameters

- **G**: NetworkX graph
- **source**: node
  - Starting node for path
- **target**: node
  - Ending node for path
- **cutoff**: integer, optional
  - Depth to stop the search. Only paths of length ≤ cutoff are returned.

Returns

- **path_generator**: generator
  - A generator that produces lists of simple paths. If there are no paths between the source and target within the given cutoff the generator produces no output.

See Also:

- all_shortest_paths, shortest_path

Notes

This algorithm uses a modified depth-first search to generate the paths [R271]. A single path can be found in \(O(V + E)\) time but the number of simple paths in a graph can be very large, e.g. \(O(n!)\) in the complete graph of order \(n\).

References

[R271]

Examples

```python
>>> G = nx.complete_graph(4)
>>> for path in nx.all_simple_paths(G, source=0, target=3):
...     print(path)
... ```
4.33 Swap

Swap edges in a graph.

**double_edge_swap**

```
double_edge_swap(G[, nswap, max_tries])
```

Swap two edges in the graph while keeping the node degrees fixed.

A double-edge swap removes two randomly chosen edges u-v and x-y and creates the new edges u-x and v-y:

\[
\begin{align*}
\text{u--v} & \quad \text{becomes} \quad | |
\text{x--y} & \quad | \\
\text{u} & \quad \text{v} \\
\text{x} & \quad \text{y}
\end{align*}
\]

If either the edge u-x or v-y already exist no swap is performed and another attempt is made to find a suitable edge pair.

**Parameters**

- **G**: graph
  - An undirected graph
- **nswap**: integer (optional, default=1)
  - Number of double-edge swaps to perform
- **max_tries**: integer (optional)
  - Maximum number of attempts to swap edges

**Returns**

- **G**: graph
  - The graph after double edge swaps.

**Notes**

Does not enforce any connectivity constraints.

The graph G is modified in place.
4.33.2 connected_double_edge_swap

**connected_double_edge_swap** \((G, n\text{swap}=1)\)

Attempt nswap double-edge swaps in the graph G.

A double-edge swap removes two randomly chosen edges \(u-v\) and \(x-y\) and creates the new edges \(u-x\) and \(v-y\):

\[
\begin{align*}
  u -- v & \quad \text{becomes} \quad | \\
  x -- y & \quad \text{becomes} \quad | 
\end{align*}
\]

If either the edge \(u-x\) or \(v-y\) already exist no swap is performed so the actual count of swapped edges is always \(\leq n\text{swap}\).

**Parameters**

- **G** : graph
  - An undirected graph
- **n\text{swap}** : integer (optional, default=1)
  - Number of double-edge swaps to perform

**Returns**

- **G** : int
  - The number of successful swaps

**Notes**

The initial graph G must be connected, and the resulting graph is connected. The graph G is modified in place.

**References**

[R272]

4.34 Traversal

4.34.1 Depth First Search

Basic algorithms for depth-first searching.


<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>dfs_edges</strong> ((G, \text{source}))</td>
<td>Produce edges in a depth-first-search (DFS).</td>
</tr>
<tr>
<td><strong>dfs_tree</strong> ((G, \text{source}))</td>
<td>Return oriented tree constructed from a depth-first-search from source.</td>
</tr>
<tr>
<td><strong>dfs_predecessors</strong> ((G, \text{source}))</td>
<td>Return dictionary of predecessors in depth-first-search from source.</td>
</tr>
<tr>
<td><strong>dfs_successors</strong> ((G, \text{source}))</td>
<td>Return dictionary of successors in depth-first-search from source.</td>
</tr>
<tr>
<td><strong>dfs_preorder_nodes</strong> ((G, \text{source}))</td>
<td>Produce nodes in a depth-first-search pre-ordering starting from source.</td>
</tr>
<tr>
<td><strong>dfs_postorder_nodes</strong> ((G, \text{source}))</td>
<td>Produce nodes in a depth-first-search post-ordering starting from source.</td>
</tr>
<tr>
<td><strong>dfs_labeled_edges</strong> ((G, \text{source}))</td>
<td>Produce edges in a depth-first-search (DFS) labeled by type.</td>
</tr>
</tbody>
</table>

**dfs_edges**

**dfs_edges** \((G, \text{source}=\text{None})\)

Produce edges in a depth-first-search (DFS).
Parameters  

**G**: NetworkX graph  

**source**: node, optional  

Specify starting node for depth-first search and return edges in the component reachable from source.  

Returns  

**edges**: generator  

A generator of edges in the depth-first-search.

Notes  


If a source is not specified then a source is chosen arbitrarily and repeatedly until all components in the graph are searched.

Examples  

```python  
>>> G = nx.Graph()  
>>> G.add_path([0,1,2])  
>>> print(list(nx.dfs_edges(G,0)))  
[(0, 1), (1, 2)]  
```

dfs_tree  

**dfs_tree** 

`(G, source)`  

Return oriented tree constructed from a depth-first-search from source.

Parameters  

**G**: NetworkX graph  

**source**: node, optional  

Specify starting node for depth-first search.  

Returns  

**T**: NetworkX DiGraph  

An oriented tree

Examples  

```python  
>>> G = nx.Graph()  
>>> G.add_path([0,1,2])  
>>> T = nx.dfs_tree(G,0)  
>>> print(T.edges())  
[(0, 1), (1, 2)]  
```

dfs_predecessors  

**dfs_predecessors**  

`(G, source=None)`  

Return dictionary of predecessors in depth-first-search from source.

Parameters  

**G**: NetworkX graph  

**source**: node, optional
Specify starting node for depth-first search and return edges in the component reachable from source.

**Returns**  
`pred`: dict  
A dictionary with nodes as keys and predecessor nodes as values.

**Notes**


If a source is not specified then a source is chosen arbitrarily and repeatedly until all components in the graph are searched.

**Examples**

```python
>>> G = nx.Graph()
>>> G.add_path([0,1,2])
>>> print(nx.dfs_predecessors(G, 0))
{1: 0, 2: 1}
```

**dfs_successors**

**dfs_successors** *(G, source=None)*  
Return dictionary of successors in depth-first-search from source.

**Parameters**  
`G`: NetworkX graph  
`source`: node, optional  
Specify starting node for depth-first search and return edges in the component reachable from source.

**Returns**  
`succ`: dict  
A dictionary with nodes as keys and list of successor nodes as values.

**Notes**


If a source is not specified then a source is chosen arbitrarily and repeatedly until all components in the graph are searched.

**Examples**

```python
>>> G = nx.Graph()
>>> G.add_path([0,1,2])
>>> print(nx.dfs_successors(G, 0))
{0: [1], 1: [2]}
```
**dfs_preorder_nodes**

**dfs_preorder_nodes** *(G, source=None)*

Produce nodes in a depth-first-search pre-ordering starting from source.

**Parameters**

- **G**: NetworkX graph
- **source**: node, optional
  
  Specify starting node for depth-first search and return edges in the component reachable from source.

**Returns**

- **nodes**: generator
  
  A generator of nodes in a depth-first-search pre-ordering.

**Notes**


If a source is not specified then a source is chosen arbitrarily and repeatedly until all components in the graph are searched.

**Examples**

```python
>>> G = nx.Graph()
>>> G.add_path([0,1,2])
>>> print(list(nx.dfs_preorder_nodes(G,0)))
[0, 1, 2]
```

**dfs_postorder_nodes**

**dfs_postorder_nodes** *(G, source=None)*

Produce nodes in a depth-first-search post-ordering starting from source.

**Parameters**

- **G**: NetworkX graph
- **source**: node, optional
  
  Specify starting node for depth-first search and return edges in the component reachable from source.

**Returns**

- **nodes**: generator
  
  A generator of nodes in a depth-first-search post-ordering.

**Notes**


If a source is not specified then a source is chosen arbitrarily and repeatedly until all components in the graph are searched.
Examples

```python
>>> G = nx.Graph()
>>> G.add_path([0,1,2])
>>> print(list(nx.dfs_postorder_nodes(G,0)))
[2, 1, 0]
```

dfs_labeled_edges

dfs_labeled_edges \((G, source=\text{None})\)

Produce edges in a depth-first-search (DFS) labeled by type.

Parameters  

- `G`: NetworkX graph
- `source`: node, optional

Specify starting node for depth-first search and return edges in the component reachable from source.

Returns  

- `edges`: generator

A generator of edges in the depth-first-search labeled with `‘forward’, ‘nontree’, and ‘reverse’`.

Notes


If a source is not specified then a source is chosen arbitrarily and repeatedly until all components in the graph are searched.

Examples

```python
>>> G = nx.Graph()
>>> G.add_path([0,1,2])
>>> edges = (list(nx.dfs_labeled_edges(G,0)))
```

4.34.2 Breadth First Search

Basic algorithms for breadth-first searching.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bfs_edges</code></td>
<td>Produce edges in a breadth-first-search starting at source.</td>
</tr>
<tr>
<td><code>bfs_tree</code></td>
<td>Return an oriented tree constructed from of a breadth-first-search starting at source.</td>
</tr>
<tr>
<td><code>bfs_predecessors</code></td>
<td>Return dictionary of predecessors in breadth-first-search from source.</td>
</tr>
<tr>
<td><code>bfs_successors</code></td>
<td>Return dictionary of successors in breadth-first-search from source.</td>
</tr>
</tbody>
</table>

bfs_edges

bfs_edges \((G, source, reverse=False)\)

Produce edges in a breadth-first-search starting at source.

Parameters  

- `G`: NetworkX graph
source : node, optional

Specify starting node for breadth-first search and return edges in the component reachable from source.

reverse : bool, optional

If True traverse a directed graph in the reverse direction

Returns  edges : generator

A generator of edges in the breadth-first-search.

Notes


If a source is not specified then a source is chosen arbitrarily and repeatedly until all components in the graph are searched.

Examples

```python
>>> G = nx.Graph()
>>> G.add_path([0,1,2])
>>> print(list(nx.bfs_edges(G,0)))
[(0, 1), (1, 2)]
```

bfs_tree

bfs_tree(G, source, reverse=False)

Return an oriented tree constructed from of a breadth-first-search starting at source.

Parameters  

G : NetworkX graph

source : node, optional

Specify starting node for breadth-first search and return edges in the component reachable from source.

reverse : bool, optional

If True traverse a directed graph in the reverse direction

Returns  T : NetworkX DiGraph

An oriented tree

Notes


If a source is not specified then a source is chosen arbitrarily and repeatedly until all components in the graph are searched.
Examples

```python
>>> G = nx.Graph()
>>> G.add_path([0,1,2])
>>> print(list(nx.bfs_edges(G,0)))
[(0, 1), (1, 2)]
```

**bfs_predecessors**

*bfs_predecessors* (*G, source*)

Return dictionary of predecessors in breadth-first-search from source.

**Parameters**

- `G` : NetworkX graph
- `source` : node, optional
  
  Specify starting node for breadth-first search and return edges in the component reachable from source.

**Returns**

- `pred` : dict
  
  A dictionary with nodes as keys and predecessor nodes as values.

**Notes**


If a source is not specified then a source is chosen arbitrarily and repeatedly until all components in the graph are searched.

**Examples**

```python
>>> G = nx.Graph()
>>> G.add_path([0,1,2])
>>> print(nx.bfs_predecessors(G,0))
{1: 0, 2: 1}
```

**bfs_successors**

*bfs_successors* (*G, source*)

Return dictionary of successors in breadth-first-search from source.

**Parameters**

- `G` : NetworkX graph
- `source` : node, optional
  
  Specify starting node for breadth-first search and return edges in the component reachable from source.

**Returns**

- `succ` : dict
  
  A dictionary with nodes as keys and list of successors nodes as values.
Notes


If a source is not specified then a source is chosen arbitrarily and repeatedly until all components in the graph are searched.

Examples

```python
>>> G = nx.Graph()
>>> G.add_path([0,1,2])
>>> print(nx.bfs_successors(G,0))
{0: [1], 1: [2]}
```

4.35 Tree

4.35.1 Recognition

Recognition Tests

A forest is an acyclic, undirected graph, and a tree is a connected forest. Depending on the subfield, there are various conventions for generalizing these definitions to directed graphs.

In one convention, directed variants of forest and tree are defined in an identical manner, except that the direction of the edges is ignored. In effect, each directed edge is treated as a single undirected edge. Then, additional restrictions are imposed to define branchings and arborescences.

In another convention, directed variants of forest and tree correspond to the previous convention’s branchings and arborescences, respectively. Then two new terms, polyforest and polytree, are defined to correspond to the other convention’s forest and tree.

Summarizing:

<table>
<thead>
<tr>
<th>Convention 1</th>
<th>Convention 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>forest</td>
<td>polyforest</td>
</tr>
<tr>
<td>tree</td>
<td>polytree</td>
</tr>
<tr>
<td>branching</td>
<td>forest</td>
</tr>
<tr>
<td>arborescence</td>
<td>tree</td>
</tr>
</tbody>
</table>

Each convention has its reasons. The first convention emphasizes definitional similarity in that directed forests and trees are only concerned with acyclicity and do not have an in-degree constraint, just as their undirected counterparts do not. The second convention emphasizes functional similarity in the sense that the directed analog of a spanning tree is a spanning arborescence. That is, take any spanning tree and choose one node as the root. Then every edge is assigned a direction such there is a directed path from the root to every other node. The result is a spanning arborescence.

NetworkX follows the first convention. Explicitly, these are:

- **undirected forest** An undirected graph with no undirected cycles.
- **undirected tree** A connected, undirected forest.
- **directed forest** A directed graph with no undirected cycles. Equivalently, the underlying graph structure (which ignores edge orientations) is an undirected forest. In another convention, this is known as a polyforest.

4.35. Tree 335
**directed tree**  A weakly connected, directed forest. Equivalently, the underlying graph structure (which ignores edge orientations) is an undirected tree. In another convention, this is known as a polytree.

**branching**  A directed forest with each node having, at most, one parent. So the maximum in-degree is equal to 1. In another convention, this is known as a forest.

**arborescence**  A directed tree with each node having, at most, one parent. So the maximum in-degree is equal to 1. In another convention, this is known as a tree.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>is_tree(G)</td>
<td>Returns True if G is a tree.</td>
</tr>
<tr>
<td>is_forest(G)</td>
<td>Returns True if G is a forest.</td>
</tr>
</tbody>
</table>

**is_tree**

**is_tree** *(G)*

Returns *True* if *G* is a tree.

A tree is a connected graph with no undirected cycles.

For directed graphs, *G* is a tree if the underlying graph is a tree. The underlying graph is obtained by treating each directed edge as a single undirected edge in a multigraph.

**Parameters**  

*G* : graph  

The graph to test.

**Returns**  

*b* : bool  

A boolean that is *True* if *G* is a tree.

**See Also:**  

is_arborescence

**Notes**

In another convention, a directed tree is known as a *polytree* and then tree corresponds to an *arborescence*.

**is_forest**

**is_forest** *(G)*

Returns *True* if *G* is a forest.

A forest is a graph with no undirected cycles.

For directed graphs, *G* is a forest if the underlying graph is a forest. The underlying graph is obtained by treating each directed edge as a single undirected edge in a multigraph.

**Parameters**  

*G* : graph  

The graph to test.

**Returns**  

*b* : bool  

A boolean that is *True* if *G* is a forest.

**See Also:**  

is_branching
Notes

In another convention, a directed forest is known as a polyforest and then forest corresponds to a branching.

4.36 Vitality

Vitality measures.

\[ \text{closeness_vitality}(G[, \text{weight}]) \quad \text{Compute closeness vitality for nodes.} \]

4.36.1 closeness_vitality

closeness_vitality\( (G, \text{weight}=\text{None}) \)

Compute closeness vitality for nodes.

Closeness vitality of a node is the change in the sum of distances between all node pairs when excluding that node.

Parameters

- **G**: graph
- **weight**: None or string (optional)

The name of the edge attribute used as weight. If None the edge weights are ignored.

Returns

- **nodes**: dictionary

Dictionary with nodes as keys and closeness vitality as the value.

See Also:

closeness_centrality

References

[R273]

Examples

```python
>>> G=nx.cycle_graph(3)
>>> nx.closeness_vitality(G)
{0: 4.0, 1: 4.0, 2: 4.0}
```
FUNCTIONS

Functional interface to graph methods and assorted utilities.

5.1 Graph

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>degree(G[, nbunch, weight])</code></td>
<td>Return degree of single node or of nbunch of nodes.</td>
</tr>
<tr>
<td><code>degree_histogram(G)</code></td>
<td>Return a list of the frequency of each degree value.</td>
</tr>
<tr>
<td><code>density(G)</code></td>
<td>Return the density of a graph.</td>
</tr>
<tr>
<td><code>info(G[, n])</code></td>
<td>Print short summary of information for the graph G or the node n.</td>
</tr>
<tr>
<td><code>create_empty_copy(G[, with_nodes])</code></td>
<td>Return a copy of the graph G with all of the edges removed.</td>
</tr>
<tr>
<td><code>is_directed(G)</code></td>
<td>Return True if graph is directed.</td>
</tr>
</tbody>
</table>

5.1.1 degree

degree(G, nbunch=None, weight=None)

Return degree of single node or of nbunch of nodes. If nbunch is ommitted, then return degrees of all nodes.

5.1.2 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))
5.1.3 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 a graph without edges and 1 for a complete graph. The density of multigraphs can be higher than 1. Self loops are counted in the total number of edges so graphs with self loops can have density higher than 1.

5.1.4 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 \( G \)

5.1.5 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.
5.1.6 is_directed

`is_directed(G)`

Return True if graph is directed.

5.2 Nodes

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nodes(G)</code></td>
<td>Return a copy of the graph nodes in a list.</td>
</tr>
<tr>
<td><code>number_of_nodes(G)</code></td>
<td>Return the number of nodes in the graph.</td>
</tr>
<tr>
<td><code>nodes_iter(G)</code></td>
<td>Return an iterator over the graph nodes.</td>
</tr>
<tr>
<td><code>all_neighbors(graph, node)</code></td>
<td>Returns all of the neighbors of a node in the graph.</td>
</tr>
<tr>
<td><code>non_neighbors(graph, node)</code></td>
<td>Returns the non-neighbors of the node in the graph.</td>
</tr>
<tr>
<td><code>common_neighbors(G, u, v)</code></td>
<td>Return the common neighbors of two nodes in a graph.</td>
</tr>
</tbody>
</table>

5.2.1 nodes

`nodes(G)`

Return a copy of the graph nodes in a list.

5.2.2 number_of_nodes

`number_of_nodes(G)`

Return the number of nodes in the graph.

5.2.3 nodes_iter

`nodes_iter(G)`

Return an iterator over the graph nodes.

5.2.4 all_neighbors

`all_neighbors(graph, node)`

Returns all of the neighbors of a node in the graph.

If the graph is directed returns predecessors as well as successors.

Parameters

- `graph` : NetworkX graph
  - Graph to find neighbors.
- `node` : node
  - The node whose neighbors will be returned.

Returns

- `neighbors` : iterator
  - Iterator of neighbors
5.2.5 non_neighbors

**non_neighbors** *(graph, node)*

Returns the non-neighbors of the node in the graph.

**Parameters**
- **graph** : NetworkX graph
  - Graph to find neighbors.
- **node** : node
  - The node whose neighbors will be returned.

**Returns**
- **non_neighbors** : iterator
  - Iterator of nodes in the graph that are not neighbors of the node.

5.2.6 common_neighbors

**common_neighbors** *(G, u, v)*

Return the common neighbors of two nodes in a graph.

**Parameters**
- **G** : graph
  - A NetworkX undirected graph.
- **u, v** : nodes
  - Nodes in the graph.

**Returns**
- **cnbors** : iterator
  - Iterator of common neighbors of u and v in the graph.

**Raises**
- NetworkXError
  - If u or v is not a node in the graph.

Examples

```python
>>> G = nx.complete_graph(5)
>>> sorted(nx.common_neighbors(G, 0, 1))
[2, 3, 4]
```

5.3 Edges

**edges** *(G[, nbunch])* Return list of edges adjacent to nodes in nbunch.

**number_of_edges** *(G)* Return the number of edges in the graph.

**edges_iter** *(G[, nbunch])* Return iterator over edges adjacent to nodes in nbunch.

**non_edges** *(graph)* Returns the non-existent edges in the graph.

5.3.1 edges

**edges** *(G, nbunch=\text{None})*

Return list of edges adjacent to nodes in nbunch.
Return all edges if nbunch is unspecified or nbunch=None.
For digraphs, edges=out_edges

5.3.2 number_of_edges

number_of_edges \((G)\)  
Return the number of edges in the graph.

5.3.3 edges_iter

edges_iter \((G, nbunch=None)\)  
Return iterator over edges adjacent to nodes in nbunch.
Return all edges if nbunch is unspecified or nbunch=None.
For digraphs, edges=out_edges

5.3.4 non_edges

non_edges \((\text{graph})\)  
Returns the non-existent edges in the graph.

Parameters  
\text{graph} : NetworkX graph.  
Graph to find non-existent edges.

Returns  
\text{non_edges} : iterator  
Iterator of edges that are not in the graph.

5.4 Attributes

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_node_attributes ((G, name, values))</td>
<td>Set node attributes from dictionary of nodes and values</td>
</tr>
<tr>
<td>get_node_attributes ((G, name))</td>
<td>Get node attributes from graph</td>
</tr>
<tr>
<td>set_edge_attributes ((G, name, values))</td>
<td>Set edge attributes from dictionary of edge tuples and values.</td>
</tr>
<tr>
<td>get_edge_attributes ((G, name))</td>
<td>Get edge attributes from graph</td>
</tr>
</tbody>
</table>

5.4.1 set_node_attributes

set_node_attributes \((G, name, values)\)  
Set node attributes from dictionary of nodes and values

Parameters  
\text{G} : NetworkX Graph  
\text{name} : string  
Attribute name  
\text{values} : dict  
Dictionary of attribute values keyed by node. If \text{values} is not a dictionary, then it is treated as a single attribute value that is then applied to every node in \text{G}.  

5.4. Attributes
Examples

```python
>>> G = nx.path_graph(3)
>>> bb = nx.betweenness_centrality(G)
>>> nx.set_node_attributes(G, 'betweenness', bb)
>>> G.node[1]['betweenness']
1.0
```

5.4.2 get_node_attributes

`get_node_attributes(G, name)`

Get node attributes from graph

**Parameters**

- `G` : NetworkX Graph
- `name` : string
  
  Attribute name

**Returns**

Dictionary of attributes keyed by node.

Examples

```python
>>> G=nx.Graph()
>>> G.add_nodes_from([1,2,3],color='red')
>>> color=nx.get_node_attributes(G,'color')
>>> color[1]
'red'
```

5.4.3 set_edge_attributes

`set_edge_attributes(G, name, values)`

Set edge attributes from dictionary of edge tuples and values.

**Parameters**

- `G` : NetworkX Graph
- `name` : string
- `values` : dict
  
  Dictionary of attribute values keyed by edge (tuple). For multigraphs, the keys tuples must be of the form (u, v, key). For non-multigraphs, the keys must be tuples of the form (u, v). If `values` is not a dictionary, then it is treated as a single attribute value that is then applied to every edge in `G`.

Examples

```python
>>> G = nx.path_graph(3)
>>> bb = nx.edge_betweenness_centrality(G, normalized=False)
>>> nx.set_edge_attributes(G, 'betweenness', bb)
>>> G[1][2]['betweenness']
2.0
```
5.4.4 get_edge_attributes

get_edge_attributes \((G, \text{name})\)
Get edge attributes from graph

Parameters

\(G\) : NetworkX Graph

\(\text{name}\) : string

Attribute name

Returns

Dictionary of attributes keyed by edge. For (di)graphs, the keys are 2-tuples of the form: \((u,v)\). For multi(di)graphs, the keys are 3-tuples of the form: \((u, v, key)\).

Examples

```python
g=G=nx.Graph()
G.add_path([1,2,3],color='red')
color=nx.get_edge_attributes(G,'color')
color[(1,2)]
'red'
```

5.5 Freezing graph structure

freeze \((G)\)
Modify graph to prevent further change by adding or removing nodes or edges.

is_frozen \((G)\)
Return True if graph is frozen.

5.5.1 freeze

freeze \((G)\)
Modify graph to prevent further change by adding or removing nodes or edges.

Node and edge data can still be modified.

Parameters

\(G\) : graph

A NetworkX graph

See Also:

is_frozen

Notes

To “unfreeze” a graph you must make a copy by creating a new graph object:

```python
g=graph = nx.path_graph(4)
frozen_graph = nx.freeze(graph)
unfrozen_graph = nx.Graph(frozen_graph)
nx.is_frozen(unfrozen_graph)
False
```
Examples

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

5.5.2 is_frozen

`is_frozen(G)`
Return True if graph is frozen.

Parameters

G : graph
A NetworkX graph

See Also:

freeze
6.1 Atlas

Generators for the small graph atlas.


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

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

### 6.1.1 graph_atlas_g

\texttt{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.

\begin{itemize}
  \item The graphs are listed:
  \begin{enumerate}
    \item in increasing order of number of nodes;
    \item for a fixed number of nodes, in increasing order of the number of edges;
    \item for fixed numbers of nodes and edges, in increasing order of the degree sequence, for example 111223 < 112222;
    \item for fixed degree sequence, in increasing number of automorphisms.
  \end{enumerate}
\end{itemize}

Note that indexing is set up so that for \(GAG=\text{graph\_atlas\_g()}\), then \(G123=GAG[123]\) and \(G[0]=\text{empty\_graph}(0)\)

6.2 Classic

Generators for some classic graphs.

The typical graph generator is called as follows:

\texttt{>>> 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).

\texttt{balanced\_tree(r, h[, create\_using])}  
Return the perfectly balanced \(r\)-tree of height \(h\).
Table 6.2 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>barbell_graph(m1, m2[, create_using])</code></td>
<td>Return the Barbell Graph: two complete graphs connected by a path.</td>
</tr>
<tr>
<td><code>complete_graph(n[, create_using])</code></td>
<td>Return the complete graph ( K_n ) with ( n ) nodes.</td>
</tr>
<tr>
<td><code>complete_bipartite_graph(n1, n2[, create_using])</code></td>
<td>Return the complete bipartite graph ( K_{n1 \times n2} ).</td>
</tr>
<tr>
<td><code>circular_ladder_graph(n[, create_using])</code></td>
<td>Return the circular ladder graph ( CL_n ) of length ( n ).</td>
</tr>
<tr>
<td><code>cycle_graph(n[, create_using])</code></td>
<td>Return the cycle graph ( C_n ) over ( n ) nodes.</td>
</tr>
<tr>
<td><code>dorogovtsev_goltsev_mendes_graph(n[, ...])</code></td>
<td>Return the hierarchically constructed Dorogovtsev-Goltsev-Mendes graph.</td>
</tr>
<tr>
<td><code>empty_graph([n, create_using])</code></td>
<td>Return the empty graph with ( n ) nodes and zero edges.</td>
</tr>
<tr>
<td><code>grid_2d_graph(m, n[, periodic, create_using])</code></td>
<td>Return the 2d grid graph of ( mxn ) nodes, each connected to its nearest neighbors.</td>
</tr>
<tr>
<td><code>grid_graph(dim[, periodic])</code></td>
<td>Return the ( n )-dimensional grid graph.</td>
</tr>
<tr>
<td><code>hypercube_graph(n)</code></td>
<td>Return the ( n )-dimensional hypercube.</td>
</tr>
<tr>
<td><code>ladder_graph(n[, create_using])</code></td>
<td>Return the Ladder graph of length ( n ).</td>
</tr>
<tr>
<td><code>lollipop_graph(m, n[, create_using])</code></td>
<td>Return the Lollipop Graph; ( K_m ) connected to ( P_n ).</td>
</tr>
<tr>
<td><code>null_graph([create_using])</code></td>
<td>Return the Null graph with no nodes or edges.</td>
</tr>
<tr>
<td><code>path_graph(n[, create_using])</code></td>
<td>Return the Path graph ( P_n ) of ( n ) nodes linearly connected by ( n-1 ) edges.</td>
</tr>
<tr>
<td><code>star_graph(n[, create_using])</code></td>
<td>Return the Star graph with ( n+1 ) nodes: one center node, connected to all ( n ) other nodes.</td>
</tr>
<tr>
<td><code>trivial_graph([create_using])</code></td>
<td>Return the Trivial graph with one node (with integer label 0) and no edges.</td>
</tr>
<tr>
<td><code>wheel_graph(n[, create_using])</code></td>
<td>Return the Wheel graph: a single hub node connected to each node of the cycle.</td>
</tr>
</tbody>
</table>

### 6.2.1 balanced_tree

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

Return the perfectly balanced \( r \)-tree of height \( h \).

**Parameters**

- \( r \) : int
  - Branching factor of the tree
- \( h \) : int
  - Height of the tree
- create_using : NetworkX graph type, optional
  - Use specified type to construct graph (default = networkx.Graph)

**Returns**

- \( G \) : networkx Graph
  - A tree with \( n \) nodes

**Notes**

This is the rooted tree where all leaves are at distance \( h \) from the root. The root has degree \( r \) and all other internal nodes have degree \( r+1 \).

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

Also referred to as a complete \( r \)-ary tree.

### 6.2.2 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 $2m_1+m_2$ nodes are numbered $0,...,m_1-1$ for the left barbell, $m_1,...,m_1+m_2-1$ for the path, and $m_1+m_2,...,2m_1+m_2-1$ for the right barbell.

The 3 subgraphs are joined via the edges $(m_1-1,m_1)$ and $(m_1+m_2-1,m_1+m_2)$. If $m_2=0$, 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 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 complete_bipartite_graph

**complete_bipartite_graph**(*n1*, *n2*, *create_using=None*)

Return the complete bipartite graph $K_{n_1,n_2}$.

Composed of two partitions with $n_1$ nodes in the first and $n_2$ nodes in the second. Each node in the first is connected to each node in the second.

Node labels are the integers 0 to $n_1+n_2-1$.

6.2.5 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 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 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 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,

```python
>>> 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 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 grid_graph

**grid_graph** (*dim, periodic=False*)

Return the n-dimensional grid graph.

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

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

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

6.2.11 hypercube_graph

**hypercube_graph** (*n*)

Return the *n*-dimensional hypercube.

Node labels are the integers 0 to $2^{**n} - 1$. 
6.2.12 ladder_graph

\texttt{ladder_graph}(n, \texttt{create}\_\texttt{using}=\texttt{None})
\begin{itemize}
    \item Return the Ladder graph of length \( n \).
    \item This is two rows of \( n \) nodes, with each pair connected by a single edge.
    \item Node labels are the integers \( 0 \) to \( 2^*n - 1 \).
\end{itemize}

6.2.13 lollipop_graph

\texttt{lollipop_graph}(m, n, \texttt{create}\_\texttt{using}=\texttt{None})
\begin{itemize}
    \item Return the Lollipop Graph; \( K_m \) connected to \( P_n \).
    \item This is the Barbell Graph without the right barbell.
    \item For \( m>1 \) and \( n\geq 0 \), the complete graph \( K_m \) is connected to the path \( P_n \). The resulting \( m+n \) nodes are labelled \( 0,\ldots,m-1 \) for the complete graph and \( m,\ldots,m+n-1 \) for the path. The 2 subgraphs are joined via the edge \( (m-1,m) \).
    \item If \( n=0 \), this is merely a complete graph.
    \item Node labels are the integers \( 0 \) to \texttt{number\_of\_nodes} - \( 1 \).
\end{itemize}
\begin{itemize}
    \item (This graph is an extremal example in David Aldous and Jim Fill’s etext on Random Walks on Graphs.)
\end{itemize}

6.2.14 null_graph

\texttt{null_graph}(\texttt{create}\_\texttt{using}=\texttt{None})
\begin{itemize}
    \item Return the Null graph with no nodes or edges.
\end{itemize}
\begin{itemize}
    \item See empty\_graph for the use of \texttt{create}\_\texttt{using}.
\end{itemize}

6.2.15 path_graph

\texttt{path_graph}(n, \texttt{create}\_\texttt{using}=\texttt{None})
\begin{itemize}
    \item Return the Path graph \( P_n \) of \( n \) nodes linearly connected by \( n-1 \) edges.
\end{itemize}
\begin{itemize}
    \item Node labels are the integers \( 0 \) to \( n - 1 \). If \texttt{create}\_\texttt{using} is a DiGraph then the edges are directed in increasing order.
\end{itemize}

6.2.16 star_graph

\texttt{star_graph}(n, \texttt{create}\_\texttt{using}=\texttt{None})
\begin{itemize}
    \item Return the Star graph with \( n+1 \) nodes: one center node, connected to \( n \) outer nodes.
\end{itemize}
\begin{itemize}
    \item Node labels are the integers \( 0 \) to \( n \).
\end{itemize}

6.2.17 trivial_graph

\texttt{trivial_graph}(\texttt{create}\_\texttt{using}=\texttt{None})
\begin{itemize}
    \item Return the Trivial graph with one node (with integer label 0) and no edges.
\end{itemize}
6.2.18 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.

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

6.3.1 make_small_graph

make_small_graph \((graph\_description[, ...])\)
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 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](http://mathworld.wolfram.com/LCFNotation.html) for a description and references.

6.3.3 bull_graph

**bull_graph** (create_using=None)

Return the Bull graph.

6.3.4 chvatal_graph

**chvatal_graph** (create_using=None)

Return the Chvátal graph.

6.3.5 cubical_graph

**cubical_graph** (create_using=None)

Return the 3-regular Platonic Cubical graph.
6.3.6 desargues_graph

```
desargues_graph(create_using=None)
```
Return the Desargues graph.

6.3.7 diamond_graph

```
diamond_graph(create_using=None)
```
Return the Diamond graph.

6.3.8 dodecahedral_graph

```
dodecahedral_graph(create_using=None)
```
Return the Platonic Dodecahedral graph.

6.3.9 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 heawood_graph

```
heawood_graph(create_using=None)
```
Return the Heawood graph, a (3,6) cage.

6.3.11 house_graph

```
house_graph(create_using=None)
```
Return the House graph (square with triangle on top).

6.3.12 house_x_graph

```
house_x_graph(create_using=None)
```
Return the House graph with a cross inside the house square.

6.3.13 icosahedral_graph

```
icosahedral_graph(create_using=None)
```
Return the Platonic Icosahedral graph.
6.3.14 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 moebius_kantor_graph

moebius_kantor_graph(create_using=None)

Return the Moebius-Kantor graph.

6.3.16 octahedral_graph

octahedral_graph(create_using=None)

Return the Platonic Octahedral graph.

6.3.17 pappus_graph

pappus_graph()

Return the Pappus graph.

6.3.18 petersen_graph

petersen_graph(create_using=None)

Return the Petersen graph.

6.3.19 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 tetrahedral_graph

tetrahedral_graph(create_using=None)

Return the 3-regular Platonic Tetrahedral graph.

6.3.21 truncated_cube_graph

truncated_cube_graph(create_using=None)

Return the skeleton of the truncated cube.
6.3.22 truncated_tetrahedron_graph

*truncated_tetrahedron_graph*(create_using=None)
Return the skeleton of the truncated Platonic tetrahedron.

6.3.23 tutte_graph

tutte_graph*(create_using=None)*
Return the Tutte graph.

6.4 Random Graphs

Generators for random graphs.

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

6.4.1 fast_gnp_random_graph

*fast_gnp_random_graph*(n, p, seed=None, directed=False)
Return a random graph $G_{n,p}$ (Erdős-Rényi graph, binomial graph).

**Parameters**

- **n** : int
  The number of nodes.
- **p** : float
  Probability for edge creation.
- **seed** : int, optional
  Seed for random number generator (default=None).
- **directed** : bool, optional (default=False)
  If True return a directed graph

**See Also:**

gnp_random_graph
Notes

The \( G_{n,p} \) graph algorithm chooses each of the \([n(n-1)]/2 \) (undirected) or \( n(n-1) \) (directed) possible edges with probability \( p \).

This algorithm is \( O(n+m) \) where \( m \) is the expected number of edges \( m=p*n*(n-1)/2 \).

It should be faster than \texttt{gnp_random_graph} when \( p \) is small and the expected number of edges is small (sparse graph).

References

[R306]

6.4.2 \texttt{gnp_random_graph}

\texttt{gnp_random_graph}(n, p, seed=None, directed=False)

Return a random graph \( G_{n,p} \) (Erdős-Rényi graph, binomial graph).

Chooses each of the possible edges with probability \( p \).

This is also called \texttt{binomial_graph} and \texttt{erdos_renyi_graph}.

Parameters

\begin{itemize}
  \item \texttt{n} : int
    \text{The number of nodes.}
  \item \texttt{p} : float
    \text{Probability for edge creation.}
  \item \texttt{seed} : int, optional
    \text{Seed for random number generator (default=None).}
  \item \texttt{directed} : bool, optional (default=False)
    \text{If True return a directed graph}
\end{itemize}

See Also:

\texttt{fast_gnp_random_graph}

Notes

This is an \( O(n^2) \) algorithm. For sparse graphs (small \( p \)) see \texttt{fast_gnp_random_graph} for a faster algorithm.

References

[R307], [R308]
6.4.3 dense_gnm_random_graph

dense_gnm_random_graph \((n, m, seed=None)\)

Return the random graph \(G_{n,m}\).

Gives a graph picked randomly out of the set of all graphs with \(n\) nodes and \(m\) edges. This algorithm should be faster than gnm_random_graph for dense graphs.

Parameters

\(n\) : int

The number of nodes.

\(m\) : int

The number of edges.

\(seed\) : int, optional

Seed for random number generator (default=None).

See Also:

gnm_random_graph

Notes

Algorithm by Keith M. Briggs Mar 31, 2006. Inspired by Knuth’s Algorithm S (Selection sampling technique), in section 3.4.2 of [R303].

References

[R303]

6.4.4 gnm_random_graph

gnm_random_graph \((n, m, seed=None, directed=False)\)

Return the random graph \(G_{n,m}\).

Produces a graph picked randomly out of the set of all graphs with \(n\) nodes and \(m\) edges.

Parameters

\(n\) : int

The number of nodes.

\(m\) : int

The number of edges.

\(seed\) : int, optional

Seed for random number generator (default=None).

\(directed\) : bool, optional (default=False)

If True return a directed graph
6.4.5 erdos_renyi_graph

```python
erdos_renyi_graph(n, p, seed=None, directed=False)
```

Return a random graph \( G_{n,p} \) (Erdős-Rényi graph, binomial graph).

Chooses each of the possible edges with probability \( p \).

This is also called binomial_graph and erdos_renyi_graph.

**Parameters**

- \( n : \text{int} \)
  - The number of nodes.
- \( p : \text{float} \)
  - Probability for edge creation.
- \( \text{seed} : \text{int, optional} \)
  - Seed for random number generator (default=None).
- \( \text{directed} : \text{bool, optional (default=False)} \)
  - If True return a directed graph

**See Also:**

- `fast_gnp_random_graph`

**Notes**

This is an \( O(n^2) \) algorithm. For sparse graphs (small \( p \)) see `fast_gnp_random_graph` for a faster algorithm.

**References**

[R304], [R305]

6.4.6 binomial_graph

```python
binomial_graph(n, p, seed=None, directed=False)
```

Return a random graph \( G_{n,p} \) (Erdős-Rényi graph, binomial graph).

Chooses each of the possible edges with probability \( p \).

This is also called binomial_graph and erdos_renyi_graph.

**Parameters**

- \( n : \text{int} \)
  - The number of nodes.
- \( p : \text{float} \)
  - Probability for edge creation.
- \( \text{seed} : \text{int, optional} \)
  - Seed for random number generator (default=None).
- \( \text{directed} : \text{bool, optional (default=False)} \)
  - If True return a directed graph
See Also:

fast_gnp_random_graph

Notes

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

References

[R301], [R302]

6.4.7 newman_watts_strogatz_graph

newman_watts_strogatz_graph (n, k, p, 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

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

[R309]

6.4.8 watts_strogatz_graph

watts_strogatz_graph (n, k, p, 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

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

[R313]

6.4.9 connected_watts_strogatz_graph

connected_watts_strogatz_graph (n, k, p, tries=100, seed=None)
   Return a connected Watts-Strogatz small-world graph.

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

   Parameters

   n : int
      The number of nodes

   k : int
      Each node is connected to k nearest neighbors in ring topology

   p : float
      The probability of rewiring each edge

   tries : int
      Number of attempts to generate a connected graph.

   seed : int, optional
      The seed for random number generator.
See Also:

newman_watts_strogatz_graph, watts_strogatz_graph

6.4.10 random_regular_graph

random_regular_graph(d, n, seed=None)

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

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

Parameters

- d : int
  Degree
- n : integer
  Number of nodes. The value of n*d must be even.
- seed : hashable object
  The seed for random number generator.

Notes

The nodes are numbered from 0 to n-1.

Kim and Vu’s paper [R312] shows that this algorithm samples in an asymptotically uniform way from the space of random graphs when \( d = O(n^{1/3-\epsilon}) \).

References

[R311], [R312]

6.4.11 barabasi_albert_graph

barabasi_albert_graph(n, m, seed=None)

Return random graph using Barabási-Albert preferential attachment model.

A graph of n nodes is grown by attaching new nodes each with m edges that are preferentially attached to existing nodes with high degree.

Parameters

- n : int
  Number of nodes
- m : int
  Number of edges to attach from a new node to existing nodes
- seed : 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

[R300]

6.4.12 powerlaw_cluster_graph

\texttt{powerlaw\_cluster\_graph}(n, m, p, seed=None)

Holme and Kim algorithm for growing graphs with powerlaw degree distribution and approximate average clustering.

**Parameters**
- \( n \) : 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
- \( seed \) : int, optional
  - Seed for random number generator (default=\text{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

[R310]

6.4.13 random_lobster

\texttt{random\_lobster}(n, p1, p2, 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

\( p_1 \) : float

Probability of adding an edge to the backbone

\( p_2 \) : float

Probability of adding an edge one level beyond backbone

\( \text{seed} \) : int, optional

Seed for random number generator (default=None).

### 6.4.14 random_shell_graph

random_shell_graph\((\text{constructor, seed=None})\)

Return a random shell graph for the constructor given.

**Parameters**  
\( \text{constructor} \): a list of three-tuples

\( (n,m,d) \) for each shell starting at the center shell.

\( n \) : int

The number of nodes in the shell

\( m \) : int

The number of edges in the shell

\( d \) : float

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

\( \text{seed} \) : int, optional

Seed for random number generator (default=None).

**Examples**

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

### 6.4.15 random_powerlaw_tree

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

Return a tree with a powerlaw degree distribution.

**Parameters**  
\( n \) : int,

The number of nodes

\( \text{gamma} \) : float

Exponent of the power-law

\( \text{seed} \) : int, optional

```
Seed for random number generator (default=None).

tries : int

Number of attempts to adjust sequence to make a tree

Notes

A trial powerlaw degree sequence is chosen and then elements are swapped with new elements from a powerlaw distribution until the sequence makes a tree (#edges=#nodes-1).

6.4.16 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[, seed, selfloops]) Return a random graph with given expected degrees.

havel_hakimi_graph(deg_sequence[, create_using]) Return a simple graph with given degree sequence constructed
directed_havel_hakimi_graph(in_deg_sequence, ...)

degree_sequence_tree(deg_sequence[, ...]) Make a tree for the given degree sequence.

random_degree_sequence_graph(sequence[, ...]) Return a simple random graph with the given degree sequence.

6.5.1 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 [R277].

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.

The density of self-loops and parallel edges tends to decrease as the number of nodes increases. However, typically the number of self-loops will approach a Poisson distribution with a nonzero mean, and similarly for the number of parallel edges. Consider a node with k stubs. The probability of being joined to another stub of the same node is basically \((k-1)/N\) where k is the degree and N is the number of nodes. So the probability of a self-loop scales like \(c/N\) for some constant c. As N grows, this means we expect c self-loops. Similarly for parallel edges.

**References**

[R277]

**Examples**

```python
>>> from networkx.utils import powerlaw_sequence
>>> z=nx.utils.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 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 [R278].
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

[R278]

6.5. Degree Sequence
Examples

```python
>>> 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:

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

To remove self loops:

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

### 6.5.3 expected_degree_graph

**expected_degree_graph** *(w, seed=None, selfloops=True)*

Return a random graph with given expected degrees.

Given a sequence of expected degrees \( W = (w_0, w_1, \ldots, w_{n-1}) \) of length \( n \) this algorithm assigns an edge between node \( u \) and node \( v \) with probability

\[
p_{uv} = \frac{w_u w_v}{\sum_k w_k}.
\]

**Parameters**

- **w** : list
  
  The list of expected degrees.

- **selfloops** : bool (default=True)
  
  Set to False to remove the possibility of self-loop edges.

- **seed** : hashable object, optional
  
  The seed for the random number generator.

**Returns**

Graph

**Notes**

The nodes have integer labels corresponding to index of expected degrees input sequence.

The complexity of this algorithm is \( O(n + m) \) where \( n \) is the number of nodes and \( m \) is the expected number of edges.

The model in [R280] includes the possibility of self-loop edges. Set selfloops=False to produce a graph without self loops.

For finite graphs this model doesn’t produce exactly the given expected degree sequence. Instead the expected degrees are as follows.

For the case without self loops (selfloops=False),

\[
E[deg(u)] = \sum_{v \neq u} p_{uv} = w_u \left( 1 - \frac{w_u}{\sum_k w_k} \right).
\]
NetworkX uses the standard convention that a self-loop edge counts 2 in the degree of a node, so with self loops (selfloops=True),
\[
E[\text{deg}(u)] = \sum_{v \neq u} p_{uv} + 2p_{uu} = w_u \left( 1 + \frac{w_u}{\sum_k w_k} \right).
\]

References

[R280], [R281]

Examples

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

6.5.4 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. 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.

The basic algorithm is from Hakimi [R282] and was generalized by Kleitman and Wang [R283].

References

[R282], [R283]

6.5.5 directed_havel_hakimi_graph

**directed_havel_hakimi_graph** *(in_deg_sequence, out_deg_sequence, create_using=None)*

Return a directed graph with the given degree sequences.

**Parameters**

- **in_deg_sequence**: list of integers
  
  Each list entry corresponds to the in-degree of a node.
out_deg_sequence : list of integers
    Each list entry corresponds to the out-degree of a node.
create_using : graph, optional (default DiGraph)
    Return graph of this type. The instance will be cleared.
Returns  G : DiGraph
    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 are not digraphical.
See Also:
configuration_model

Notes
    Algorithm as described by Kleitman and Wang [R279].
References
[R279]

6.5.6 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

6.5.7 random_degree_sequence_graph

random_degree_sequence_graph (sequence, seed=None, tries=10)
    Return a simple random graph with the given degree sequence.
    If the maximum degree $d_m$ in the sequence is $O(m^{1/4})$ then the algorithm produces almost uniform random
    graphs in $O(md_m)$ time where $m$ is the number of edges.
Parameters  sequence : list of integers
    Sequence of degrees
seed : hashable object, optional
    Seed for random number generator
tries : int, optional
    Maximum number of tries to create a graph
Returns  G : Graph
    A graph with the specified degree sequence. Nodes are labeled starting at 0 with an
    index corresponding to the position in the sequence.
Raisenetworkxunfeasible

If the degree sequence is not graphical.

Networkxerror

If a graph is not produced in specified number of tries

See Also:
is_valid_degree_sequence, configuration_model

Notes

The generator algorithm [R284] is not guaranteed to produce a graph.

References

[R284]

Examples

>>> sequence = [1, 2, 2, 3]
>>> G = nx.random_degree_sequence_graph(sequence)
>>> sorted(G.degree().values())
[1, 2, 2, 3]

6.6 Random Clustered

Generate graphs with given degree and triangle sequence.

random_clustered_graph(joint_degree_sequence) Generate a random graph with the given joint degree and triangle degree sequence.

6.6.1 random_clustered_graph

random_clustered_graph(joint_degree_sequence, create_using=None, seed=None)

Generate a random graph with the given joint degree and triangle degree sequence.

This uses a configuration model-like approach to generate a random pseudograph (graph with parallel edges and self loops) by randomly assigning edges to match the given independent edge and triangle degree sequence.

Parameters joint_degree_sequence : list of integer pairs

Each list entry corresponds to the independent edge degree and triangle degree of a node.

create_using : graph, optional (default MultiGraph)

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

seed : hashable object, optional

The seed for the random number generator.

Returns G : MultiGraph

6.6. Random Clustered
A graph with the specified degree sequence. Nodes are labeled starting at 0 with an index corresponding to the position in `deg_sequence`.

**Raises** NetworkXError

If the independent edge degree sequence sum is not even or the triangle degree sequence sum is not divisible by 3.

**Notes**

As described by Miller [R298] (see also Newman [R299] for an equivalent description).

A non-graphical degree sequence (not realizable by some simple graph) is allowed since this function returns graphs with self loops and parallel edges. An exception is raised if the independent degree sequence does not have an even sum or the triangle degree sequence sum is not divisible by 3.

This configuration model-like construction process can lead to duplicate edges and loops. You can remove the self-loops and parallel edges (see below) which will likely result in a graph that doesn’t have the exact degree sequence specified. This “finite-size effect” decreases as the size of the graph increases.

**References**

[R298], [R299]

**Examples**

```python
>>> deg_tri=[[1,0],[1,0],[1,0],[2,0],[1,0],[2,1],[0,1],[0,1]]
>>> G = nx.random_clustered_graph(deg_tri)
```

To remove parallel edges:

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

To remove self loops:

```python
>>> G.remove_edges_from(G.selfloop_edges())
```

### 6.7 Directed

Generators for some directed graphs.

gn_graph: growing network gnc_graph: growing network with copying gnr_graph: growing network with redirection

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gn_graph(n[, kernel, create_using, seed])</code></td>
<td>Return the GN digraph with n nodes.</td>
</tr>
<tr>
<td><code>gnr_graph(n, p[, create_using, seed])</code></td>
<td>Return the GNR digraph with n nodes and redirection probability p.</td>
</tr>
<tr>
<td><code>gnc_graph(n[, create_using, seed])</code></td>
<td>Return the GNC digraph with n nodes.</td>
</tr>
<tr>
<td><code>scale_free_graph(n[, alpha, beta, gamma, ...])</code></td>
<td>Return a scale free directed graph.</td>
</tr>
</tbody>
</table>
### 6.7.1 gn_graph

**gn_graph** \( (n, \text{kernel}=\text{None}, \text{create\_using}=\text{None}, \text{seed}=\text{None}) \)

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**

[R285]

**Examples**

```python
>>> D=nx.gn_graph(10)  # the GN graph
>>> G=D.to_undirected() # the undirected version
```

To specify an attachment kernel use the kernel keyword

```python
>>> D=nx.gn_graph(10, kernel=lambda x:x**1.5) # A_k=k^1.5
```

### 6.7.2 gnr_graph

**gnr_graph** \( (n, p, \text{create\_using}=\text{None}, \text{seed}=\text{None}) \)

Return the GNR digraph with \( n \) nodes and redirection probability \( p \).

The GNR (growing network with redirection) graph is built by adding nodes one at a time with a link to one previously added node. The previous target node is chosen uniformly at random. With probability \( p \) the link is instead “redirected” to the successor node of the target. The graph is always a (directed) tree.

**Parameters**
- **n** : int
  - The number of nodes for the generated graph.
- **p** : float
  - The redirection probability.
- **create\_using** : graph, optional (default DiGraph)
  - Return graph of this type. The instance will be cleared.
seed : hashable object, optional
The seed for the random number generator.

References

[R287]

Examples

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

6.7.3 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

[R286]

6.7.4 scale_free_graph

scale_free_graph \((n, alpha=0.41, beta=0.54, gamma=0.05, delta\_in=0.2, delta\_out=0, create\_using=None, seed=None)\)
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 ac-
  cording 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 connected to an existing node chosen randomly according to the out-degree distribution.

delta_in : float
    Bias for choosing nodes from in-degree distribution.

delta_out : float
    Bias for choosing nodes from out-degree distribution.

create_using : graph, optional (default MultiDiGraph)
    Use this graph instance to start the process (default=3-cycle).

seed : integer, optional
    Seed for random number generator

Notes

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

References

[R288]

Examples

>>> G=nx.scale_free_graph(100)

6.8 Geometric

Generators for geometric graphs.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>random_geometric_graph(n, radius[, dim, pos])</td>
<td>Return the random geometric graph in the unit cube.</td>
</tr>
<tr>
<td>geographical_threshold_graph(n, theta[, ...])</td>
<td>Return a geographical threshold graph.</td>
</tr>
<tr>
<td>waxman_graph(n[, alpha, beta, L, domain])</td>
<td>Return a Waxman random graph.</td>
</tr>
<tr>
<td>navigable_small_world_graph(n[, p, q, r, ...])</td>
<td>Return a navigable small-world graph.</td>
</tr>
</tbody>
</table>

6.8.1 random_geometric_graph

random_geometric_graph (n, radius, dim=2, pos=None)
    Return the random geometric graph in the unit cube.

    The random geometric graph model places n nodes uniformly at random in the unit cube. Two nodes u, v are connected with an edge if \( d(u, v) \leq r \) where \( d \) is the Euclidean distance and \( r \) is a radius threshold.

    Parameters n : int
        Number of nodes
**radius**: float

Distance threshold value

**dim**: int, optional

Dimension of graph

**pos**: dict, optional

A dictionary keyed by node with node positions as values.

**Returns**

Graph

**Notes**

This uses an \( n^2 \) algorithm to build the graph. A faster algorithm is possible using k-d trees.

The pos keyword can be used to specify node positions so you can create an arbitrary distribution and domain for positions. If you need a distance function other than Euclidean you'll have to hack the algorithm.

E.g to use a 2d Gaussian distribution of node positions with mean (0,0) and std. dev. 2

```python
>>> import random
>>> n=20
>>> p=dict((i,(random.gauss(0,2),random.gauss(0,2))) for i in range(n))
>>> G = nx.random_geometric_graph(n,0.2,pos=p)
```

**References**

[R292]

**Examples**

```python
>>> G = nx.random_geometric_graph(20,0.1)
```

### 6.8.2 geographical_threshold_graph

**geographical_threshold_graph** *(n, theta, alpha=2, dim=2, pos=None, weight=None)*

Return a geographical threshold graph.

The geographical threshold graph model places \( n \) nodes uniformly at random in a rectangular domain. Each node \( u \) is assigned a weight \( w_u \). Two nodes \( u, v \) are connected with an edge if

\[
w_u + w_v \geq \theta r^\alpha
\]

where \( r \) is the Euclidean distance between \( u \) and \( v \), and \( \theta, \alpha \) are parameters.

**Parameters**

- **n**: int
  
  Number of nodes

- **theta**: float
  
  Threshold value

- **alpha**: float, optional
Exponent of distance function

dim : int, optional
    Dimension of graph

pos : dict
    Node positions as a dictionary of tuples keyed by node.

weight : dict
    Node weights as a dictionary of numbers keyed by node.

Returns  Graph

Notes

If weights are not specified they are assigned to nodes by drawing randomly from an the exponential distribution
with rate parameter $\lambda = 1$. To specify a weights from a different distribution assign them to a dictionary and
pass it as the weight= keyword

>>> import random

>>> n = 20

>>> w=dict((i,random.expovariate(5.0)) for i in range(n))

>>> G = nx.geographical_threshold_graph(20,50,weight=w)

If node positions are not specified they are randomly assigned from the uniform distribution.

References

[R289], [R290]

Examples

>>> G = nx.geographical_threshold_graph(20,50)

6.8.3 waxman_graph

waxman_graph  \((n, alpha=0.4, beta=0.1, L=None, domain=(0, 0, 1, 1))\)

Return a Waxman random graph.

The Waxman random graph models place \(n\) nodes uniformly at random in a rectangular domain. Two nodes \(u,v\)
are connected with an edge with probability

\[ p = \alpha \cdot exp(-d/(\beta \cdot L)). \]

This function implements both Waxman models.

**Waxman-1: \(L\) not specified**  The distance \(d\) is the Euclidean distance between the nodes \(u\) and \(v\). \(L\) is the
maximum distance between all nodes in the graph.

**Waxman-2: \(L\) specified**  The distance \(d\) is chosen randomly in \([0, L]\).

Parameters  

- n : int
Number of nodes

**alpha**: float

Model parameter

**beta**: float

Model parameter

**L**: float, optional

Maximum distance between nodes. If not specified the actual distance is calculated.

**domain**: tuple of numbers, optional

Domain size (xmin, ymin, xmax, ymax)

Returns

G: Graph

References

[R293]

### 6.8.4 navigable_small_world_graph

*navigable_small_world_graph*(n, p=1, q=1, r=2, dim=2, seed=None)

Return a navigable small-world graph.

A navigable small-world graph is a directed grid with additional long-range connections that are chosen randomly. From [R291]:

Begin with a set of nodes that are identified with the set of lattice points in an \( n \times n \) square, \((i, j) : i \in 1, 2, \ldots, n, j \in 1, 2, \ldots, n\) and define the lattice distance between two nodes \((i, j)\) and \((k, l)\) to be the number of “lattice steps” separating them:

\[
d((i, j), (k, l)) = |k - i| + |l - j|.
\]

For a universal constant \(p\), the node \(u\) has a directed edge to every other node within lattice distance \(p\) (local contacts).

For universal constants \(q \geq 0\) and \(r \geq 0\) construct directed edges from \(u\) to \(q\) other nodes (long-range contacts) using independent random trials; the \(i\)th directed edge from \(u\) has endpoint \(v\) with probability proportional to \(d((i, j), (k, l))^{-r}\).

**Parameters**

**n**: int

The number of nodes.

**p**: int

The diameter of short range connections. Each node is connected to every other node within lattice distance \(p\).

**q**: int

The number of long-range connections for each node.

**r**: float

Exponent for decaying probability of connections. The probability of connecting to a node at lattice distance \(d\) is \(1/d^r\).

**dim**: int

Dimension of grid
**seed** : int, optional

Seed for random number generator (default=None).

**References**

[R291]

### 6.9 Hybrid

**Hybrid**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>kl_connected_subgraph(G, k, l[, low_memory, ...])</code></td>
<td>Returns the maximum locally (k,l) connected subgraph of G.</td>
</tr>
<tr>
<td><code>is_kl_connected(G, k, l[, low_memory])</code></td>
<td>Returns True if G is kl connected.</td>
</tr>
</tbody>
</table>

#### 6.9.1 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

#### 6.9.2 is_kl_connected

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

Returns True if G is kl connected.

### 6.10 Bipartite

Generators and functions for bipartite graphs.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bipartite_configuration_model(aseq, bseq[, ...])</code></td>
<td>Return a random bipartite graph from two given degree sequences.</td>
</tr>
<tr>
<td><code>bipartite_havel_hakimi_graph(aseq, bseq[, ...])</code></td>
<td>Return a bipartite graph from two given degree sequences using Havel Hakimi.</td>
</tr>
<tr>
<td><code>bipartite_reverse_havel_hakimi_graph(aseq, bseq)</code></td>
<td>Return a bipartite graph from two given degree sequences using Havel Hakimi.</td>
</tr>
<tr>
<td><code>bipartite_alternating_havel_hakimi_graph(...)</code></td>
<td>Return a bipartite graph from two given degree sequences using Havel Hakimi.</td>
</tr>
<tr>
<td><code>bipartite_preferential_attachment_graph(aseq, p)</code></td>
<td>Create a bipartite graph with a preferential attachment model from degree sequences.</td>
</tr>
<tr>
<td><code>bipartite_random_graph(n, m, p[, seed, directed])</code></td>
<td>Return a bipartite random graph.</td>
</tr>
<tr>
<td><code>bipartite_gnmk_random_graph(n, m, k[, seed, ...])</code></td>
<td>Return a random bipartite graph G_{n,m,k}.</td>
</tr>
</tbody>
</table>

#### 6.10.1 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
Degree sequence for node set A.

bseq : list
Degree sequence for node set B.

create_using : NetworkX graph instance, optional
Return graph of this type.

seed : integer, optional
Seed for random number generator.

Nodes from the set A are connected to nodes in the set B by choosing randomly from the possible free stubs, one in A and one in B.

Notes

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

The nodes are assigned the attribute ‘bipartite’ with the value 0 or 1 to indicate which bipartite set the node belongs to.

6.10.2 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.

Nodes from the set A are connected to nodes in the set B by connecting the highest degree nodes in set A to the highest degree nodes in set B until all stubs are connected.

Parameters  

aseq : list
Degree sequence for node set A.

bseq : list
Degree sequence for node set B.

create_using : NetworkX graph instance, optional
Return graph of this type.

Notes

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

The nodes are assigned the attribute ‘bipartite’ with the value 0 or 1 to indicate which bipartite set the node belongs to.
6.10.3 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.

Nodes from set A are connected to nodes in the set B by connecting the highest degree nodes in set A to the lowest degree nodes in set B until all stubs are connected.

**Parameters**

- **aseq** : list
  Degree sequence for node set A.

- **bseq** : list
  Degree sequence for node set B.

- **create_using** : NetworkX graph instance, optional
  Return graph of this type.

**Notes**

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

The nodes are assigned the attribute ‘bipartite’ with the value 0 or 1 to indicate which bipartite set the node belongs to.

6.10.4 bipartite_alternating_havel_hakimi_graph

**bipartite_alternating_havel_hakimi_graph** *(aseq, bseq, create_using=None)*

Return a bipartite graph from two given degree sequences using an alternating Havel-Hakimi style construction.

Nodes from the set A are connected to nodes in the set B by connecting the highest degree nodes in set A to alternatively the highest and the lowest degree nodes in set B until all stubs are connected.

**Parameters**

- **aseq** : list
  Degree sequence for node set A.

- **bseq** : list
  Degree sequence for node set B.

- **create_using** : NetworkX graph instance, optional
  Return graph of this type.

**Notes**

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

The nodes are assigned the attribute ‘bipartite’ with the value 0 or 1 to indicate which bipartite set the node belongs to.
6.10.5 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
  Degree sequence for node set A.
- **p** : float
  Probability that a new bottom node is added.
- **create_using** : NetworkX graph instance, optional
  Return graph of this type.
- **seed** : integer, optional
  Seed for random number generator.

**References**
[R275]

6.10.6 bipartite_random_graph

bipartite_random_graph (n, m, p, seed=None, directed=False)
Return a bipartite random graph.
This is a bipartite version of the binomial (Erdős-Rényi) graph.

**Parameters**
- **n** : int
  The number of nodes in the first bipartite set.
- **m** : int
  The number of nodes in the second bipartite set.
- **p** : float
  Probability for edge creation.
- **seed** : int, optional
  Seed for random number generator (default=None).
- **directed** : bool, optional (default=False)
  If True return a directed graph

**See Also:**
gnp_random_graph, bipartite_configuration_model

**Notes**
The bipartite random graph algorithm chooses each of the n*m (undirected) or 2*nm (directed) possible edges with probability p.
This algorithm is O(n+m) where m is the expected number of edges.
The nodes are assigned the attribute ‘bipartite’ with the value 0 or 1 to indicate which bipartite set the node belongs to.

References

[R276]

6.10.7 bipartite_gnmk_random_graph

*bipartite_gnmk_random_graph*(n, m, k, seed=None, directed=False)

Return a random bipartite graph \(G_{n,m,k}\).

Produces a bipartite graph chosen randomly out of the set of all graphs with \(n\) top nodes, \(m\) bottom nodes, and \(k\) edges.

**Parameters**

- **n**: int
  - The number of nodes in the first bipartite set.
- **m**: int
  - The number of nodes in the second bipartite set.
- **k**: int
  - The number of edges
- **seed**: int, optional
  - Seed for random number generator (default=None).
- **directed**: bool, optional (default=False)
  - If True return a directed graph

**See Also:**

*gnm_random_graph*

**Notes**

If \(k > m \times n\) then a complete bipartite graph is returned.

This graph is a bipartite version of the \(G_{nm}\) random graph model.

**Examples**

\[
G = nx.bipartite_gnmk_random_graph(10,20,50)
\]

6.11 Line Graph

Line graph algorithms.
6.11.1 Undirected Graphs

For an undirected graph G without multiple edges, each edge can be written as a set \{u,v\}. Its line graph L has the edges of G as its nodes. If x and y are two nodes in L, then \{x,y\} is an edge in L if and only if the intersection of x and y is nonempty. Thus, the set of all edges is determined by the set of all pair-wise intersections of edges in G.

Trivially, every edge x={u,v} in G would have a nonzero intersection with itself, and so every node in L should have a self-loop. This is not so interesting, and the original context of line graphs was with simple graphs, which had no self-loops or multiple edges. The line graph was also meant to be simple graph and thus, self-loops in L are not part of the standard definition of a line graph. In a pair-wise intersection matrix, this is analogous to not including the diagonal as part of the line graph definition.

Self-loops and multiple edges in G add nodes to L in a natural way, and do not require any fundamental changes to the definition. It might be argued that the self-loops we excluded before should now be included. However, the self-loops are still “trivial” in some sense and thus, are usually excluded.

6.11.2 Directed Graphs

For a directed graph G without multiple edges, each edge can be written as a tuple (u,v). Its line graph L has the edges of G as its nodes. If x=(a,b) and y=(c,d) are two nodes in L, then (x,y) is an edge in L if and only if the tail of x matches the head of y—e.g., b=c.

Due to the directed nature of the edges, it is no longer the case that every edge x=(u,y) should be connected to itself with a self-loop in L. Now, the only time self-loops arise is if G itself has a self-loop. So such self-loops are no longer “trivial” but instead, represent essential features of the topology of G. For this reason, the historical development of line digraphs is such that self-loops are included. When the graph G has multiple edges, once again only superficial changes are required to the definition.

6.11.3 References


6.11.4 line_graph

\texttt{line\_graph(G, create\_using=None)} Return the line graph of the graph or digraph G.

Parameters

- \texttt{G : graph}

  A NetworkX Graph, DiGraph, MultiGraph, or MultiDiGraph.

Returns

- \texttt{L : graph}
The line graph of $G$.

**Notes**

Graph, node, and edge data are not propagated to the new graph. For undirected graphs, the nodes in $G$ must be sortable—otherwise, the constructed line graph may not be correct.

**Examples**

```python
>>> G = nx.star_graph(3)
>>> L = nx.line_graph(G)
>>> print(sorted(L.edges()))  # makes a clique, $K_3$
[((0, 1), (0, 2)), ((0, 1), (0, 3)), ((0, 3), (0, 2))]
```

### 6.12 Ego Graph

Ego graph.

```python
ego_graph(G, n[, radius, center, ...]) Returns induced subgraph of neighbors centered at node $n$ within a given radius.
```

#### 6.12.1 ego_graph

**ego_graph** $(G, n, radius=1, center=True, undirected=False, distance=None)$

Returns induced subgraph of neighbors centered at node $n$ within a given radius.

**Parameters**

- $G$ : graph
  - A NetworkX Graph or DiGraph
- $n$ : node
  - A single node
- $radius$ : number, optional
  - Include all neighbors of distance<=$radius$ from $n$.
- $center$ : bool, optional
  - If False, do not include center node in graph
- $undirected$ : bool, optional
  - If True use both in- and out-neighbors of directed graphs.
- $distance$ : key, optional
  - Use specified edge data key as distance. For example, setting $distance$='weight' will use the edge weight to measure the distance from the node $n$.

**Notes**

For directed graphs $D$ this produces the “out” neighborhood or successors. If you want the neighborhood of predecessors first reverse the graph with $D.reverse()$. If you want both directions use the keyword argument
undirected=True.
Node, edge, and graph attributes are copied to the returned subgraph.

6.13 Stochastic

Stochastic graph.

\texttt{stochastic\_graph(G[, copy, weight])} Return a right-stochastic representation of G.

6.13.1 stochastic\_graph

\texttt{stochastic\_graph}(G, copy=True, weight='weight')

Return a right-stochastic representation of G.

A right-stochastic graph is a weighted digraph in which all of the node (out) neighbors edge weights sum to 1.

**Parameters**

- **G**: directed graph
  A NetworkX DiGraph
- **copy**: boolean, optional
  If True make a copy of the graph, otherwise modify the original graph
- **weight**: edge attribute key (optional, default='weight')
  Edge data key used for weight. If no attribute is found for an edge the edge weight is set to 1. Weights must be positive numbers.

6.14 Intersection

Generators for random intersection graphs.

\texttt{uniform\_random\_intersection\_graph}(n, m, p[, ...]) Return a uniform random intersection graph.

\texttt{k\_random\_intersection\_graph}(n, m, k) Return a intersection graph with randomly chosen attribute sets for each node.

\texttt{general\_random\_intersection\_graph}(n, m, p) Return a random intersection graph with independent probabilities for each node.

6.14.1 uniform\_random\_intersection\_graph

\texttt{uniform\_random\_intersection\_graph}(n, m, p, seed=None)

Return a uniform random intersection graph.

**Parameters**

- **n**: int
  The number of nodes in the first bipartite set (nodes)
- **m**: int
  The number of nodes in the second bipartite set (attributes)
- **p**: float
  Probability of connecting nodes between bipartite sets
- **seed**: int, optional
Seed for random number generator (default=None).

See Also:

gnp_random_graph

References

[R296], [R297]

6.14.2 k_random_intersection_graph

**k_random_intersection_graph** *(n, m, k)*

Return a intersection graph with randomly chosen attribute sets for each node that are of equal size (k).

**Parameters**

- **n**: int
  - The number of nodes in the first bipartite set (nodes)
- **m**: int
  - The number of nodes in the second bipartite set (attributes)
- **k**: float
  - Size of attribute set to assign to each node.
- **seed**: int, optional
  - Seed for random number generator (default=None).

See Also:

gnp_random_graph, uniform_random_intersection_graph

References

[R295]

6.14.3 general_random_intersection_graph

**general_random_intersection_graph** *(n, m, p)*

Return a random intersection graph with independent probabilities for connections between node and attribute sets.

**Parameters**

- **n**: int
  - The number of nodes in the first bipartite set (nodes)
- **m**: int
  - The number of nodes in the second bipartite set (attributes)
- **p**: list of floats of length *m*
  - Probabilities for connecting nodes to each attribute
- **seed**: int, optional
  - Seed for random number generator (default=None).
See Also:

gnp_random_graph, uniform_random_intersection_graph

References

[R294]

6.15 Social Networks

Famous social networks.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>karate_club_graph()</td>
<td>Return Zachary's Karate club graph.</td>
</tr>
<tr>
<td>davis_southern_women_graph()</td>
<td>Return Davis Southern women social network.</td>
</tr>
<tr>
<td>florentine_families_graph()</td>
<td>Return Florentine families graph.</td>
</tr>
</tbody>
</table>

6.15.1 karate_club_graph

karate_club_graph()

Return Zachary’s Karate club graph.

References

[R316], [R317]

6.15.2 davis_southern_women_graph

davis_southern_women_graph()

Return Davis Southern women social network.

This is a bipartite graph.

References

[R314]

6.15.3 florentine_families_graph

florentine_families_graph()

Return Florentine families graph.

References

[R315]


7.1 Graph Matrix

Adjacency matrix and incidence matrix of graphs.

\[
\text{adjacency\_matrix}(G[, \text{nodelist, weight}]) \quad \text{Return adjacency matrix of } G.
\]

\[
\text{incidence\_matrix}(G[, \text{nodelist, edgelist, ...}]) \quad \text{Return incidence matrix of } G.
\]

7.1.1 adjacency\_matrix

\[
\text{adjacency\_matrix}(G, \text{nodelist=\text{None}, weight='weight')} \quad \text{Return adjacency matrix of } G.
\]

**Parameters**

- **G**: graph
  A NetworkX graph
- **nodelist**: list, optional
  The rows and columns are ordered according to the nodes in nodelist. If nodelist is None, then the ordering is produced by G.nodes().
- **weight**: string or None, optional (default='weight')
  The edge data key used to provide each value in the matrix. If None, then each edge has weight 1.

**Returns**

- **A**: SciPy sparse matrix
  Adjacency matrix representation of G.

**See Also:**

to\_numpy\_matrix, to\_scipy\_sparse\_matrix, to\_dict\_of\_dicts

**Notes**

If you want a pure Python adjacency matrix representation try networkx.convert.to\_dict\_of\_dicts which will return a dictionary-of-dictionaries format that can be addressed as a sparse matrix.

For MultiGraph/MultiDiGraph with parallel edges the weights are summed. See to\_numpy\_matrix for other options.
The convention used for self-loop edges in graphs is to assign the diagonal matrix entry value to the edge weight attribute (or the number 1 if the edge has no weight attribute). If the alternate convention of doubling the edge weight is desired the resulting Scipy sparse matrix can be modified as follows:

```python
>>> import scipy as sp

>>> G = nx.Graph([(1,1)])

>>> A = nx.adjacency_matrix(G)

>>> print(A.todense())
[[1]]

>>> A.setdiag(A.diagonal() * 2)

>>> print(A.todense())
[[2]]
```

7.1.2 incidence_matrix

`incidence_matrix(G, nodelist=None, edgelist=None, oriented=False, weight=None)`

Return incidence matrix of G.

The incidence matrix assigns each row to a node and each column to an edge. For a standard incidence matrix a 1 appears wherever a row’s node is incident on the column’s edge. For an oriented incidence matrix each edge is assigned an orientation (arbitrarily for undirected and aligning to direction for directed). A -1 appears for the tail of an edge and 1 for the head of the edge. The elements are zero otherwise.

**Parameters**

- **G**: graph
  A NetworkX graph

- **nodelist**: list, optional (default=all nodes in G)
  The rows are ordered according to the nodes in nodelist. If nodelist is None, then the ordering is produced by G.nodes().

- **edgelist**: list, optional (default=all edges in G)
  The columns are ordered according to the edges in edgelist. If edgelist is None, then the ordering is produced by G.edges().

- **oriented**: bool, optional (default=False)
  If True, matrix elements are +1 or -1 for the head or tail node respectively of each edge. If False, +1 occurs at both nodes.

- **weight**: string or None, optional (default=None)
  The edge data key used to provide each value in the matrix. If None, then each edge has weight 1. Edge weights, if used, should be positive so that the orientation can provide the sign.

**Returns**

- **A**: SciPy sparse matrix
  The incidence matrix of G.

**Notes**

For MultiGraph/MultiDiGraph, the edges in edgelist should be (u,v,key) 3-tuples.

“Networks are the best discrete model for so many problems in applied mathematics” [R318].
References

[R318]

7.2 Laplacian Matrix

Laplacian matrix of graphs.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>laplacian_matrix(G[, nodelist, weight])</td>
<td>Return the Laplacian matrix of G.</td>
</tr>
<tr>
<td>normalized_laplacian_matrix(G[, nodelist, ...])</td>
<td>Return the normalized Laplacian matrix of G.</td>
</tr>
<tr>
<td>directed_laplacian_matrix(G[, nodelist, ...])</td>
<td>Return the directed Laplacian matrix of G.</td>
</tr>
</tbody>
</table>

7.2.1 laplacian_matrix

**laplacian_matrix** *(G, nodelist=None, weight='weight')*

Return the Laplacian matrix of G.

The graph Laplacian is the matrix \( L = D - A \), where \( A \) is the adjacency matrix and \( D \) is the diagonal matrix of node degrees.

**Parameters**

- **G**: graph
  A NetworkX graph

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

- **weight**: string or None, optional (default='weight')
  The edge data key used to compute each value in the matrix. If None, then each edge has weight 1.

**Returns**

- **L**: SciPy sparse matrix
  The Laplacian matrix of G.

**See Also**

to_numpy_matrix, normalized_laplacian_matrix

**Notes**

For MultiGraph/MultiDiGraph, the edges weights are summed.

7.2.2 normalized_laplacian_matrix

**normalized_laplacian_matrix** *(G, nodelist=None, weight='weight')*

Return the normalized Laplacian matrix of G.

The normalized graph Laplacian is the matrix

\[
NL = D^{-1/2}LD^{-1/2}
\]

where \( L \) is the graph Laplacian and \( D \) is the diagonal matrix of node degrees.
Parameters

G : graph
   A NetworkX graph

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

weight : string or None, optional (default='weight')
   The edge data key used to compute each value in the matrix. If None, then each edge has weight 1.

Returns

L : NumPy matrix
   The normalized Laplacian matrix of G.

See Also:

laplacian_matrix

Notes

For MultiGraph/MultiDiGraph, the edges weights are summed. See to_numpy_matrix for other options.
If the Graph contains selfloops, D is defined as diag(sum(A,1)), where A is the adjacency matrix [R321].

References

[R320], [R321]

7.2.3 directed_laplacian_matrix

directed_laplacian_matrix (G, nodelist=None, weight='weight', walk_type=None, alpha=0.95)

Return the directed Laplacian matrix of G.

The graph directed Laplacian is the matrix

\[ L = I - \left( \Phi^{1/2} P \Phi^{-1/2} + \Phi^{-1/2} P^T \Phi^{1/2} \right) / 2 \]

where \( I \) is the identity matrix, \( P \) is the transition matrix of the graph, and \( \Phi \) a matrix with the Perron vector of \( P \) in the diagonal and zeros elsewhere.

Depending on the value of walk_type, \( P \) can be the transition matrix induced by a random walk, a lazy random walk, or a random walk with teleportation (PageRank).

Parameters

G : DiGraph
   A NetworkX graph

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

weight : string or None, optional (default='weight')
   The edge data key used to compute each value in the matrix. If None, then each edge has weight 1.
**walk_type**: string or None, optional (default=None)

If None, \( P \) is selected depending on the properties of the graph. Otherwise is one of ‘random’, ‘lazy’, or ‘pagerank’

**alpha**: real

\((1 - \alpha)\) is the teleportation probability used with pagerank

**Returns**

L : NumPy array

Normalized Laplacian of G.

**Raises**

NetworkXError

If NumPy cannot be imported

NetworkXNotImplemented

If G is not a DiGraph

**See Also:**

laplacian_matrix

**Notes**

Only implemented for DiGraphs

**References**

[R319]

### 7.3 Spectrum

Eigenvalue spectrum of graphs.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>laplacian_spectrum(G[, weight])</td>
<td>Return eigenvalues of the Laplacian of G</td>
</tr>
<tr>
<td>adjacency_spectrum(G[, weight])</td>
<td>Return eigenvalues of the adjacency matrix of G</td>
</tr>
</tbody>
</table>

#### 7.3.1 laplacian_spectrum

**laplacian_spectrum**(G, weight='weight')

Return eigenvalues of the Laplacian of G

**Parameters**

G : graph

A NetworkX graph

weight : string or None, optional (default='weight')

The edge data key used to compute each value in the matrix. If None, then each edge has weight 1.

**Returns**

evals : NumPy array

Eigenvalues
7.3.2 adjacency_spectrum

`adjacency_spectrum(G[, weight])` Return eigenvalues of the adjacency matrix of G.

**Parameters**
- **G**: graph
  - A NetworkX graph
- **weight**: string or None, optional (default='weight')
  - The edge data key used to compute each value in the matrix. If None, then each edge has weight 1.

**Returns**
- **evals**: NumPy array
  - Eigenvalues

**See Also:**
- adjacency_matrix

**Notes**
For MultiGraph/MultiDiGraph, the edges weights are summed. See to_numpy_matrix for other options.

7.4 Algebraic Connectivity

Algebraic connectivity and Fiedler vectors of undirected graphs.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>algebraic_connectivity(G[, ...])</code></td>
<td>Return the algebraic connectivity of an undirected graph.</td>
</tr>
<tr>
<td><code>fiedler_vector(G[, ...])</code></td>
<td>Return the Fiedler vector of a connected undirected graph.</td>
</tr>
<tr>
<td><code>spectral_ordering(G[, ...])</code></td>
<td>Compute the spectral_ordering of a graph.</td>
</tr>
</tbody>
</table>

7.4.1 algebraic_connectivity

`algebraic_connectivity(G[, weight=’weight’, normalized=False, tol=1e-08, method=’tracemin’])` Return the algebraic connectivity of an undirected graph.

The algebraic connectivity of a connected undirected graph is the second smallest eigenvalue of its Laplacian matrix.

**Parameters**
- **G**: NetworkX graph
  - An undirected graph.
- **weight**: object, optional
The data key used to determine the weight of each edge. If None, then each edge has unit weight. Default value: None.

**normalized** : bool, optional

Whether the normalized Laplacian matrix is used. Default value: False.

**tol** : float, optional


**method** : string, optional

Method of eigenvalue computation. It should be one of ‘tracemin’ (TraceMIN), ‘lanczos’ (Lanczos iteration) and ‘lobpcg’ (LOBPCG). Default value: ‘tracemin’.

The TraceMIN algorithm uses a linear system solver. The following values allow specifying the solver to be used.

<table>
<thead>
<tr>
<th>Value</th>
<th>Solver</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘tracemin_pcg’</td>
<td>Preconditioned conjugate gradient method</td>
</tr>
<tr>
<td>‘tracemin_chol’</td>
<td>Cholesky factorization</td>
</tr>
<tr>
<td>‘tracemin_lu’</td>
<td>LU factorization</td>
</tr>
</tbody>
</table>

**Returns**  

algebraic_connectivity : float

Algebraic connectivity.

**Raises**  

NetworkXNotImplemented

If G is directed.

NetworkXError

If G has less than two nodes.

**See Also:**

laplacian_matrix

**Notes**

Edge weights are interpreted by their absolute values. For MultiGraph’s, weights of parallel edges are summed. Zero-weighted edges are ignored.

To use Cholesky factorization in the TraceMIN algorithm, the scikit.sparse package must be installed.

### 7.4.2 fiedler_vector

**fiedler_vector** *(G, weight='weight', normalized=False, tol=1e-08, method='tracemin')*

Return the Fiedler vector of a connected undirected graph.

The Fiedler vector of a connected undirected graph is the eigenvector corresponding to the second smallest eigenvalue of the Laplacian matrix of of the graph.

**Parameters**

- **G** : NetworkX graph
  
  An undirected graph.

- **weight** : object, optional
  
  The data key used to determine the weight of each edge. If None, then each edge has unit weight. Default value: None.
normalized : bool, optional

Whether the normalized Laplacian matrix is used. Default value: False.

tol : float, optional


method : string, optional

Method of eigenvalue computation. It should be one of ‘tracemin’ (TraceMIN), ‘lanczos’ (Lanczos iteration) and ‘lobpcg’ (LOBPCG). Default value: ‘tracemin’.

The TraceMIN algorithm uses a linear system solver. The following values allow specifying the solver to be used.

<table>
<thead>
<tr>
<th>Value</th>
<th>Solver</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘tracemin_pcg’</td>
<td>Preconditioned conjugate gradient method</td>
</tr>
<tr>
<td>‘tracemin_chol’</td>
<td>Cholesky factorization</td>
</tr>
<tr>
<td>‘tracemin_lu’</td>
<td>LU factorization</td>
</tr>
</tbody>
</table>

Returns  fiedler_vector : NumPy array of floats.

Fiedler vector.

Raises   NetworkXNotImplemented

If G is directed.

NetworkXError

If G has less than two nodes or is not connected.

See Also:

laplacian_matrix

Notes

Edge weights are interpreted by their absolute values. For MultiGraph’s, weights of parallel edges are summed. Zero-weighted edges are ignored.

To use Cholesky factorization in the TraceMIN algorithm, the scikit.sparse package must be installed.

7.4.3 spectral_ordering

spectral_ordering (G, weight='weight', normalized=False, tol=1e-08, method='tracemin')

Compute the spectral_ordering of a graph.

The spectral ordering of a graph is an ordering of its nodes where nodes in the same weakly connected components appear contiguous and ordered by their corresponding elements in the Fiedler vector of the component.

Parameters  G : NetworkX graph

A graph.

weight : object, optional

The data key used to determine the weight of each edge. If None, then each edge has unit weight. Default value: None.

normalized : bool, optional

Whether the normalized Laplacian matrix is used. Default value: False.
tol : float, optional

method : string, optional
    Method of eigenvalue computation. It should be one of ‘tracemin’ (TraceMIN), ‘lanczos’ (Lanczos iteration) and ‘lobpcg’ (LOBPCG). Default value: ‘tracemin’.

The TraceMIN algorithm uses a linear system solver. The following values allow specifying the solver to be used.

<table>
<thead>
<tr>
<th>Value</th>
<th>Solver</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘tracemin_pcg’</td>
<td>Preconditioned conjugate gradient method</td>
</tr>
<tr>
<td>‘tracemin_chol’</td>
<td>Cholesky factorization</td>
</tr>
<tr>
<td>‘tracemin_lu’</td>
<td>LU factorization</td>
</tr>
</tbody>
</table>

Returns spectral_ordering : NumPy array of floats.
    Spectral ordering of nodes.

Raises NetworkXError
    If G is empty.

See Also:
    laplacian_matrix

Notes
    Edge weights are interpreted by their absolute values. For MultiGraph’s, weights of parallel edges are summed. Zero-weighted edges are ignored.

To use Cholesky factorization in the TraceMIN algorithm, the scikit.sparse package must be installed.

### 7.5 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. |

#### 7.5.1 attr_matrix

attr_matrix(G, edge_attr=None, node_attr=None, normalized=False, rc_order=None, dtype=None, order=None)
    Returns a NumPy matrix using attributes from G.

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

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

Parameters
g : graph
The NetworkX graph used to construct the NumPy matrix.

**edge_attr** : str, optional

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

**node_attr** : str, optional

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

**normalized** : bool, optional

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

**rc_order** : list, optional

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

**Returns**

**M** : NumPy matrix

The attribute matrix.

**ordering** : list

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

**Other Parameters**

**dtype** : NumPy data-type, optional

A valid NumPy dtype used to initialize the array. Keep in mind certain dtypes can yield unexpected results if the array is to be normalized. The parameter is passed to `numpy.zeros()`. If unspecified, the NumPy default is used.

**order** : {'C', 'F'}, optional

Whether to store multidimensional data in C- or Fortran-contiguous (row- or column-wise) order in memory. This parameter is passed to `numpy.zeros()`. If unspecified, the NumPy default is used.

**Examples**

Construct an adjacency matrix:

```python
>>> 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

7.5.2 attr_sparse_matrix

def attr_sparse_matrix(G, edge_attr=None, node_attr=None, normalized=False, rc_order=None, dtype=None):
    Returns a SciPy sparse matrix using attributes from G.

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

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

    Parameters:
    G : graph
        The NetworkX graph used to construct the NumPy matrix.
    edge_attr : str, optional
        Each element of the matrix represents a running total of the specified edge attribute for edges whose node attributes correspond to the rows/cols of the matrix. The attribute must be present for all edges in the graph. If no attribute is specified, then we just count the number of edges whose node attributes correspond to the matrix element.
    node_attr : str, optional
        Node attribute to use for constructing the matrix.
Each row and column in the matrix represents a particular value of the node attribute. The attribute must be present for all nodes in the graph. Note, the values of this attribute should be reliably hashable. So, float values are not recommended. If no attribute is specified, then the rows and columns will be the nodes of the graph.

**normalized** : bool, optional

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

**rc_order** : list, optional

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

**Returns**  
M : SciPy sparse matrix

The attribute matrix.

ordering : list

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

**Other Parameters**

dtype : NumPy data-type, optional

A valid NumPy dtype used to initialize the array. Keep in mind certain dtypes can yield unexpected results if the array is to be normalized. The parameter is passed to numpy.zeros(). If unspecified, the NumPy default is used.

**Examples**

Construct an adjacency matrix:

```python
>>> 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.

```python
>>> 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)

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

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

\[
\begin{align*}
\Pr( v \text{ is red } | u \text{ is red}) &= \frac{1}{3} \\
\Pr( v \text{ is blue } | u \text{ is red}) &= \frac{2}{3} \\
\Pr( v \text{ is red } | u \text{ is blue}) &= 1 \\
\Pr( v \text{ is blue } | u \text{ is blue}) &= 0
\end{align*}
\]

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

```python
>>> M = nx.attr_sparse_matrix(G, edge_attr='weight', node_attr='color', rc_order=rc)
>>> 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
CONVERTING TO AND FROM OTHER DATA FORMATS

8.1 To NetworkX Graph

Functions to convert NetworkX graphs to and from other formats.

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

8.1.1 Examples

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

```python
>>> 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 to_networkx_graph

```python
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

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

instead of the equivalent

```python
>>> 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 multi-
graph from a multigraph.

8.2 Dictionaries

<table>
<thead>
<tr>
<th>to_dict_of_dicts(G[, nodelist, edge_data])</th>
<th>Return adjacency representation of graph as a dictionary of dictionaries.</th>
</tr>
</thead>
<tbody>
<tr>
<td>from_dict_of_dicts(d[, create_using, ...])</td>
<td>Return a graph from a dictionary of dictionaries.</td>
</tr>
</tbody>
</table>

8.2.1 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).

8.2.2 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

```python
>>> dod= {0: {1:{'weight':1}}}
# single edge (0,1)
>>> G=nx.from_dict_of_dicts(dod)
```

or

```python
>>> G=nx.Graph(dod) # use Graph constructor
```

### 8.3 Lists

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>to_dict_of_lists(G[, nodelist])</code></td>
<td>Return adjacency representation of graph as a dictionary of lists.</td>
</tr>
<tr>
<td><code>from_dict_of_lists(d[, create_using])</code></td>
<td>Return a graph from a dictionary of lists.</td>
</tr>
<tr>
<td><code>to_edgelist(G[, nodelist])</code></td>
<td>Return a list of edges in the graph.</td>
</tr>
<tr>
<td><code>from_edgelist(edgelist[, create_using])</code></td>
<td>Return a graph from a list of edges.</td>
</tr>
</tbody>
</table>

#### 8.3.1 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.

#### 8.3.2 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**

```python
>>> dol= {0:[1]}  
# single edge (0,1)
>>> G=nx.from_dict_of_lists(dol)
```

or

```python
>>> G=nx.Graph(dol)  
# use Graph constructor
```
### 8.3.3 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

### 8.3.4 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**

```python
>>> edgelist= [(0,1)]  # single edge (0,1)
>>> G=nx.from_edgelist(edgelist)
```

or

```python
>>> G=nx.Graph(edgelist) # use Graph constructor
```

### 8.4 Numpy

Functions to convert NetworkX graphs to and from numpy/scipy matrices.

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

#### 8.4.1 Examples

Create a 10 node random graph from a numpy matrix

```python
>>> import numpy

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

or equivalently

```python
>>> D = nx.to_networkx_graph(a,create_using=nx.DiGraph())
```
8.4.2 See Also

nx_pygraphviz, nx_pydot
to_numpy_matrix(G[, nodelist, dtype, order, ...])  Return the graph adjacency matrix as a NumPy matrix.

to_numpy_recarray(G[, nodelist, dtype, order])  Return the graph adjacency matrix as a NumPy recarray.

from_numpy_matrix(A[, create_using])  Return a graph from numpy matrix.

8.4.3 to_numpy_matrix

to_numpy_matrix(G, nodelist=None, dtype=None, order=None, multigraph_weight=<built-in function sum>, weight='weight', nonedge=0.0)

Return the graph adjacency matrix as a NumPy matrix.

Parameters

- **G**: graph
  - The NetworkX graph used to construct the NumPy matrix.

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

- **dtype**: NumPy data type, optional
  - A valid single NumPy data type used to initialize the array. This must be a simple type such as int or numpy.float64 and not a compound data type (see to_numpy_recarray) If None, then the NumPy default is used.

- **order**: {'C', 'F'}, optional
  - Whether to store multidimensional data in C- or Fortran-contiguous (row- or column-wise) order in memory. If None, then the NumPy default is used.

- **multigraph_weight**: {sum, min, max}, optional
  - An operator that determines how weights in multigraphs are handled. The default is to sum the weights of the multiple edges.

- **weight**: string or None optional (default='weight')
  - The edge attribute that holds the numerical value used for the edge weight. If an edge does not have that attribute, then the value 1 is used instead.

- **nonedge**: float (default=0.0)
  - The matrix values corresponding to nonedges are typically set to zero. However, this could be undesirable if there are matrix values corresponding to actual edges that also have the value zero. If so, one might prefer nonedges to have some other value, such as nan.

Returns

- **M**: NumPy matrix
  - Graph adjacency matrix

See Also:

to_numpy_recarray, from_numpy_matrix

Notes

The matrix entries are assigned to the weight edge attribute. When an edge does not have a weight attribute, the value of the entry is set to the number 1. For multiple (parallel) edges, the values of the entries are determined by the ‘multigraph_weight’ parameter. The default is to sum the weight attributes for each of the parallel edges.
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`.

The convention used for self-loop edges in graphs is to assign the diagonal matrix entry value to the weight attribute of the edge (or the number 1 if the edge has no weight attribute). If the alternate convention of doubling the edge weight is desired the resulting Numpy matrix can be modified as follows:

```python
>>> import numpy as np
>>> G = nx.Graph([(1,1)])
>>> A = nx.to_numpy_matrix(G)
>>> A
matrix([[ 1.]])
>>> A.A[np.diag_indices_from(A)] *= 2
>>> A
matrix([[ 2.]])
```

### Examples

```python
>>> G = nx.MultiDiGraph()
>>> G.add_edge(0,1,weight=2)
>>> G.add_edge(1,0)
>>> G.add_edge(2,2,weight=3)
>>> G.add_edge(2,2)
>>> nx.to_numpy_matrix(G, nodelist=[0,1,2])
matrix([[ 0., 2., 0.],
        [ 1., 0., 0.],
        [ 0., 0., 4.]])
```

### 8.4.4 `to_numpy_recarray` 

`to_numpy_recarray` \((G, nodelist=None, dtype=[('weight', <type 'float'>)], order=None)\)

Return the graph adjacency matrix as a NumPy recarray.

**Parameters**

- **G**: graph
  - The NetworkX graph used to construct the NumPy matrix.
- **nodelist**: list, optional
  - The rows and columns are ordered according to the nodes in `nodelist`. If `nodelist` is None, then the ordering is produced by `G.nodes()`.
- **dtype**: NumPy data-type, optional
  - A valid NumPy named dtype used to initialize the NumPy recarray. The data type names are assumed to be keys in the graph edge attribute dictionary.
- **order**: {'C', 'F'}, optional
  - Whether to store multidimensional data in C- or Fortran-contiguous (row- or column-wise) order in memory. If None, then the NumPy default is used.

**Returns**

- **M**: NumPy recarray
  - The graph with specified edge data as a NumPy recarray
Notes

When `nodelist` does not contain every node in \( G \), the matrix is built from the subgraph of \( G \) that is induced by the nodes in `nodelist`.

Examples

```python
>>> G = nx.Graph()
>>> G.add_edge(1,2,weight=7.0,cost=5)
>>> A=nx.to_numpy_recarray(G,dtype=[('weight',float),('cost',int)])
>>> print(A.weight)
[[-0.  7.]
 [ 7.  0.]]
>>> print(A.cost)
[[0 5]
 [5 0]]
```

8.4.5 from_numpy_matrix

**from_numpy_matrix** \( (A, create_using=None) \)

Return a graph from numpy matrix.

The numpy matrix is interpreted as an adjacency matrix for the graph.

**Parameters**

- **A** : numpy matrix
  An adjacency matrix representation of a graph

- **create_using** : NetworkX graph
  Use specified graph for result. The default is Graph()

**See Also:**

to_numpy_matrix, to_numpy_recarray

Notes

If the numpy matrix has a single data type for each matrix entry it will be converted to an appropriate Python data type.

If the numpy matrix has a user-specified compound data type the names of the data fields will be used as attribute keys in the resulting NetworkX graph.

Examples

Simple integer weights on edges:

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

User defined compound data type on edges:
>>> import numpy
>>> dt=[('weight',float),('cost',int)]
>>> A=numpy.matrix([[1.0,2]],dtype=dt)
>>> G=nx.from_numpy_matrix(A)
>>> G.edges()
[(0, 0)]
>>> G[0][0]['cost']
2
>>> G[0][0]['weight']
1.0

8.5 Scipy

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

8.5.1 to_scipy_sparse_matrix

to_scipy_sparse_matrix(G[, nodelist, dtype, ...])

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.

- **weight**: string or None optional (default='weight')
  - The edge attribute that holds the numerical value used for the edge weight. If None then all edge weights are 1.

- **format**: str in {'bsr', 'csr', 'csc', 'coo', 'lil', 'dia', 'dok'}
  - The type of the matrix to be returned (default 'csr'). For some algorithms different implementations of sparse matrices can perform better. See [R274] for details.

Returns

- **M**: SciPy sparse matrix
  - Graph adjacency matrix.

Notes

- The matrix entries are populated using the edge attribute held in parameter weight. When an edge does not have that attribute, the value of the entry is 1.

- For multiple edges the matrix values are the sums of the edge weights.
When \texttt{nodelist} does not contain every node in \( G \), the matrix is built from the subgraph of \( G \) that is induced by the nodes in \texttt{nodelist}.

Uses coo_matrix format. To convert to other formats specify the format= keyword.

The convention used for self-loop edges in graphs is to assign the diagonal matrix entry value to the weight attribute of the edge (or the number 1 if the edge has no weight attribute). If the alternate convention of doubling the edge weight is desired the resulting Scipy sparse matrix can be modified as follows:

```python
>>> import scipy as sp
>>> G = nx.Graph({(1,1)})
>>> A = nx.to_scipy_sparse_matrix(G)
>>> print(A.todense())
[[1]]
>>> A.setdiag(A.diagonal()*2)
>>> print(A.todense())
[[2]]
```

References

[R274]

Examples

```python
>>> 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])
>>> print(S.todense())
[[0 2 0]
 [1 0 0]
 [0 0 4]]
```

### 8.5.2 from_scipy_sparse_matrix

**from_scipy_sparse_matrix** \((A, \text{create\_using}=\text{None, edge\_attribute}=\text{'weight'})\)

Return a graph from scipy sparse matrix adjacency list.

- **Parameters**
  - \( A \): scipy sparse matrix
    - An adjacency matrix representation of a graph
  - \text{create\_using}: NetworkX graph
    - Use specified graph for result. The default is Graph()
  - \text{edge\_attribute}: string
    - Name of edge attribute to store matrix numeric value. The data will have the same type as the matrix entry (int, float, (real,imag)).
Examples

```python
>>> 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=None, create_using=None, nodetype=None, encoding='utf-8']) 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=None]) 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 read_adjlist

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

Read graph in adjacency list format from path.

**Parameters**

- **path** : string or file
  
  Filename or file handle to read. Filenames ending in .gz or .bz2 will be uncompressed.

- **create_using** : NetworkX graph container
  
  Use given NetworkX graph for holding nodes or edges.

- **nodetype** : Python type, optional
  
  Convert nodes to this type.
comments : string, optional
    Marker for comment lines

delimiter : string, optional
    Separator for node labels. The default is whitespace.

create_using : NetworkX graph container
    Use given NetworkX graph for holding nodes or edges.

Returns
    G : NetworkX graph
        The graph corresponding to the lines in adjacency list format.

See Also:
    write_adjlist

Notes

This format does not store graph or node data.

Examples

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

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

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

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

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

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

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

will attempt to convert all nodes to integer type.
Since nodes must be hashable, the function nodetype must return hashable types (e.g. int, float, str, frozenset -
or tuples of those, etc.)

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

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

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

**Parameters**
- \(G\) : NetworkX graph
- \(\text{path}\) : string or file
  - Filename or file handle for data output. Filenames ending in .gz or .bz2 will be compressed.
- \(\text{comments}\) : string, optional
  - Marker for comment lines
- \(\text{delimiter}\) : string, optional
  - Separator for node labels
- \(\text{encoding}\) : string, optional
  - Text encoding.

**See Also:**
- read_adjlist, generate_adjlist

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

**Examples**

```python
>>> 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.

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

9.1.4 parse_adjlist

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

**Parameters**
- \(lines\) : list or iterator of strings
  - Input data in adjlist format
- \text{create\_using} : NetworkX graph container
  - Use given NetworkX graph for holding nodes or edges.
- \text{nodetype} : Python type, optional
  - Convert nodes to this type.
- \text{comments} : string, optional
Marker for comment lines

**delimiter** : string, optional

Separator for node labels. The default is whitespace.

**create_using** : NetworkX graph container

Use given NetworkX graph for holding nodes or edges.

**Returns**  
G: NetworkX graph

The graph corresponding to the lines in adjacency list format.

**See Also:**

read_adjlist

**Examples**

```python
def main():
    lines = ['1 2 5',
             '2 3 4',
             '3 5',
             '4',
             '5']
    G = nx.parse_adjlist(lines, nodetype=int)
    print(G.nodes())
    print(G.edges())
```

**9.1.5 generate_adjlist**

genenerate_adjlist(G, delimiter=' ')

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

**Parameters**  
G : NetworkX graph

delimiter : string, optional

Separator for node labels

**Returns**  
lines : string

Lines of data in adjlist format.

**See Also:**

write_adjlist, read_adjlist

**Examples**

```python
def main():
    G = nx.lollipop_graph(4, 3)
    for line in nx.generate_adjlist(G):
        print(line)
```

418  Chapter 9. Reading and writing graphs
9.2 Multiline Adjacency List

Read and write NetworkX graphs as multi-line adjacency lists.

The multi-line adjacency list format is useful for graphs with nodes that can be meaningfully represented as strings. With this format simple edge data can be stored but node or graph data is not.

9.2.1 Format

The first label in a line is the source node label followed by the node degree d. The next d lines are target node labels and optional edge data. That pattern repeats for all nodes in the graph.

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

```plaintext
# example.multiline-adjlist
a 2
  b
  c
d 1
e
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>read_multiline_adjlist</code></td>
<td>Read graph in multi-line adjacency list format from path.</td>
</tr>
<tr>
<td><code>write_multiline_adjlist</code></td>
<td>Write the graph G in multiline adjacency list format to path.</td>
</tr>
<tr>
<td><code>parse_multiline_adjlist</code></td>
<td>Parse lines of a multiline adjacency list representation of a graph.</td>
</tr>
<tr>
<td><code>generate_multiline_adjlist</code></td>
<td>Generate a single line of the graph G in multiline adjacency list format.</td>
</tr>
</tbody>
</table>

9.2.2 read_multiline_adjlist

`read_multiline_adjlist(path[, comments, ...])`  
Read graph in multi-line adjacency list format from path.

Parameters

- **path**: string or file
  
  Filename or file handle to read. Filenames ending in .gz or .bz2 will be uncompressed.

- **create_using**: NetworkX graph container
  
  Use given NetworkX graph for holding nodes or edges.

- **nodetype**: Python type, optional
  
  Convert nodes to this type.

- **edgetype**: Python type, optional
  
  Convert edge data to this type.

- **comments**: string, optional
  
  Marker for comment lines
NetworkX Reference, Release 1.9

**delimiter**: string, optional

Separator for node labels. The default is whitespace.

**create_using**: NetworkX graph container

Use given NetworkX graph for holding nodes or edges.

**Returns**

G: NetworkX graph

**See Also**

write_multiline_adjlist

**Notes**

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

**Examples**

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

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

```python
>>> fh=open("test.adjlist", ‘rb’)
>>> G=nx.read_multiline_adjlist(fh)
```

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

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

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

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

will attempt to convert all nodes to integer type.

The optional edgetype is a function to convert edge data strings to edgetype.

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

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

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

### 9.2.3 write_multiline_adjlist

**write_multiline_adjlist** *(G, path, delimiter=’ ‘, comments=’#’, encoding=’utf-8’)*

Write the graph G in multiline adjacency list format to path

**Parameters**

G : NetworkX graph

comments : string, optional
Marker for comment lines

**delimiter**: string, optional

Separator for node labels

**encoding**: string, optional

Text encoding.

**See Also:**

*read_multiline_adjlist*

**Examples**

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

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

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

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

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

### 9.2.4 parse_multiline_adjlist

**parse_multiline_adjlist** *(lines, comments='#', delimiter=None, create_using=None, nodetype=None, edgetype=None)*

Parse lines of a multiline adjacency list representation of a graph.

**Parameters**

- **lines**: list or iterator of strings
  
  Input data in multiline adjlist format

- **create_using**: NetworkX graph container
  
  Use given NetworkX graph for holding nodes or edges.

- **nodetype**: Python type, optional
  
  Convert nodes to this type.

- **comments**: string, optional
  
  Marker for comment lines

- **delimiter**: string, optional
  
  Separator for node labels. The default is whitespace.

**create_using**: NetworkX graph container

Use given NetworkX graph for holding nodes or edges.

**Returns**

- **G**: NetworkX graph
  
  The graph corresponding to the lines in multiline adjacency list format.
Examples

```python
>>> lines = ['1 2',
...          '2 {weight:3, name: 'Frodo'},
...          '3 {}',
...          '2 1',
...          '5 {weight:6, name: 'Saruman'}',
]

>>> G = nx.parse_multiline_adjlist(iter(lines), nodetype = int)

>>> G.nodes()
[1, 2, 3, 5]
```

9.2.5 generate_multiline_adjlist

generate_multiline_adjlist \((G, \text{delimiter}=')\)
Generate a single line of the graph \(G\) in multiline adjacency list format.

**Parameters**

- \(G\) : NetworkX graph
  - \(delimiter\) : string, optional
    - Separator for node labels

**Returns**

- \(lines\) : string
  - Lines of data in multiline adjlist format.

**See Also:**

- write_multiline_adjlist, read_multiline_adjlist

Examples

```python
>>> G = nx.lollipop_graph(4, 3)

>>> for line in nx.generate_multiline_adjlist(G):
...     print(line)
0 3
1 {}
2 {}
3 {}
1 2
2 {}
3 {}
2 1
3 {}
3 1
4 {}
4 1
5 {}
5 1
6 {}
6 0
```

9.3 Edge List

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

### 9.3.1 Format

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

Node pairs with no data:

```
1 2
```

Python dictionary as data:

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

Arbitrary data:

```
1 2 7 green
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_edgelist(path[, comments, delimiter, ...])</td>
<td>Read a graph from a list of edges.</td>
</tr>
<tr>
<td>write_edgelist(G, path[, comments, ...])</td>
<td>Write graph as a list of edges.</td>
</tr>
<tr>
<td>read_weighted_edgelist(path[, comments, ...])</td>
<td>Read a graph as list of edges with numeric weights.</td>
</tr>
<tr>
<td>write_weighted_edgelist(G, path[, comments, ...])</td>
<td>Write graph G as a list of edges with numeric weights.</td>
</tr>
<tr>
<td>generate_edgelist(G[, delimiter, data])</td>
<td>Generate a single line of the graph G in edge list format.</td>
</tr>
<tr>
<td>parse_edgelist(lines[, comments, delimiter, ...])</td>
<td>Parse lines of an edge list representation of a graph.</td>
</tr>
</tbody>
</table>

### 9.3.2 read_edgelist

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

Read a graph from a list of edges.

**Parameters**

- **path**: file or string
  - File or filename to write. If a file is provided, it must be opened in ‘rb’ mode. Filenames ending in .gz or .bz2 will be uncompressed.

- **comments**: string, optional
  - The character used to indicate the start of a comment.

- **delimiter**: string, optional
  - The string used to separate values. The default is whitespace.

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

- **nodetype**: int, float, str, Python type, optional
  - Convert node data from strings to specified type

- **data**: bool or list of (label,type) tuples
  - Tuples specifying dictionary key names and types for edge data

- **edgetype**: int, float, str, Python type, optional OBSOLETE
Convert edge data from strings to specified type and use as ‘weight’

**encoding:** string, optional

Specify which encoding to use when reading file.

**Returns**

G : graph

A networkx Graph or other type specified with create_using

See Also:

text

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

```python
>>> 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)
>>> fh.close()

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

Edgelist with data in a list:

```python
>>> textline = '1 2 3'
>>> fh = open('test.edgelist','w')
>>> d = fh.write(textline)
>>> fh.close()
>>> G = nx.read_edgelist('test.edgelist', nodetype=int, data=((’weight’,float),))
>>> G.nodes()
[1, 2]
>>> G.edges(data = True)
[(1, 2, {’weight’: 3.0})]
```

See parse_edgelist() for more examples of formatting.

### 9.3.3 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'])

9.3.4 read_weighted_edgelist

read_weighted_edgelist (path, comments='#', delimiter=None, create_using=None, nodetype=None, encoding='utf-8')
Read a graph as list of edges with numeric weights.

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

    comments : string, optional
        The character used to indicate the start of a comment.

    delimiter : string, optional
        The string used to separate values. The default is whitespace.

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

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

9.3. Edge List
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:

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

### 9.3.5 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**

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

`generate_edgelist` \((G, \text{delimiter}=' ', \text{data}=\text{True})\)
Generate a single line of the graph \(G\) in edge list format.

**Parameters**
- \(G\) : NetworkX graph
  - `delimiter` : string, optional
    Separator for node labels
  - `data` : bool or list of keys
    If False generate no edge data. If True use a dictionary representation of edge data. If a list of keys use a list of data values corresponding to the keys.

**Returns**
- `lines` : string
  Lines of data in adjlist format.

**See Also:**
- `write_adjlist`, `read_adjlist`

**Examples**

```python
>>> 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 3
1 3 {}
2 3 {}
3 4 12
4 5 {}
5 6 {}

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

9.3. Edge List
9.3.7 parse_edgelist

parse_edgelist \((lines, \text{ comments='#'}, \text{ delimiter=None}, \text{ create_using=None}, \text{ nodetype=None}, \text{ data=True})\)

Parse lines of an edge list representation of a graph.

**Parameters**
- **lines**: list or iterator of strings
  Input data in edgelist format
- **comments**: string, optional
  Marker for comment lines
- **delimiter**: string, optional
  Separator for node labels
- **create_using**: NetworkX graph container, optional
  Use given NetworkX graph for holding nodes or edges.
- **nodetype**: Python type, optional
  Convert nodes to this type.
- **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.

**Returns**
- **G**: NetworkX Graph
  The graph corresponding to lines

**See Also**
- read_weighted_edgelist

**Examples**

Edgelist with no data:

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

Edgelist with data in Python dictionary representation:
>>> lines = [
"1 2 {'weight':3}'
...
"2 3 {'weight':27}'
...
"3 4 {'weight':3.0}"
]

>>> G = nx.parse_edgelist(lines, nodetype = int)

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

>>> G.edges(data = True)
[(1, 2, {'weight': 3}), (2, 3, {'weight': 27}), (3, 4, {'weight': 3.0})]

Edgelist with data in a list:

>>> lines = [
"1 2 3",
...
"2 3 27",
...
"3 4 3.0"
]

>>> G = nx.parse_edgelist(lines, nodetype = int, data=((‘weight’,float),))

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

>>> G.edges(data = True)
[(1, 2, {'weight': 3.0}), (2, 3, {'weight': 27.0}), (3, 4, {'weight': 3.0})]

9.4 GEXF

Read and write graphs in GEXF format.

GEXF (Graph Exchange XML Format) is a language for describing complex network structures, their associated data and dynamics.

This implementation does not support mixed graphs (directed and undirected edges together).

9.4.1 Format

GEXF is an XML format. See http://gexf.net/format/schema.html for the specification and http://gexf.net/format/basic.html for examples.

`read_gexf(path[, node_type, relabel, version])` Read graph in GEXF format from path.

`write_gexf(G, path[, encoding, prettyprint, ...])` Write G in GEXF format to path.

`relabel_gexf_graph(G)` Relabel graph using "label" node keyword for node label.

9.4.2 read_gexf

`read_gexf(path, node_type=None, relabel=False, version=’1.1draft’)`

Read graph in GEXF format from path.

“GEXF (Graph Exchange XML Format) is a language for describing complex networks structures, their associated data and dynamics” [R322].

**Parameters**

- **path**: file or string
  - File or file name to write. File names ending in .gz or .bz2 will be compressed.

- **node_type**: Python type (default: None)
  - Convert node ids to this type if not None.

- **relabel**: bool (default: False)
If True relabel the nodes to use the GEXF node “label” attribute instead of the node “id” attribute as the NetworkX node label.

**Returns**

graph: NetworkX graph

If no parallel edges are found a Graph or DiGraph is returned. Otherwise a MultiGraph or MultiDiGraph is returned.

**Notes**

This implementation does not support mixed graphs (directed and undirected edges together).

**References**

[R322]

### 9.4.3 write_gexf

**write_gexf** *(G, path, encoding='utf-8', prettyprint=True, version='1.1draft')*

Write G in GEXF format to path.

“GEXF (Graph Exchange XML Format) is a language for describing complex networks structures, their associated data and dynamics” [R323].

**Parameters**

- **G**: graph
  - A NetworkX graph

- **path**: file or string
  - File or file name to write. File names ending in .gz or .bz2 will be compressed.

- **encoding**: string (optional)
  - Encoding for text data.

- **prettyprint**: bool (optional)
  - If True use line breaks and indenting in output XML.

**Notes**

This implementation does not support mixed graphs (directed and undirected edges together).

The node id attribute is set to be the string of the node label. If you want to specify an id use set it as node data, e.g. node[‘a’][‘id’]=1 to set the id of node ‘a’ to 1.

**References**

[R323]
Examples

```python
>>> G=nx.path_graph(4)
>>> nx.write_gexf(G, "test.gexf")
```

### 9.4.4 relabel_gexf_graph

**relabel_gexf_graph** *(G)*

Relabel graph using “label” node keyword for node label.

**Parameters**

- **G** : graph
  A NetworkX graph read from GEXF data

**Returns**

- **H** : graph
  A NetworkX graph with relabeled nodes

**Notes**

This function relabels the nodes in a NetworkX graph with the “label” attribute. It also handles relabeling the specific GEXF node attributes “parents”, and “pid”.

### 9.5 GML

Read graphs in GML format.

“GML, the Graph 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](http://www.infosun.fim.uni-passau.de/Graphlet/GML/gml-tr.html)

Requires pyparsing: [http://pyparsing.wikispaces.com/](http://pyparsing.wikispaces.com/)

#### 9.5.1 Format

See [http://www.infosun.fim.uni-passau.de/Graphlet/GML/gml-tr.html](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/](http://www-personal.umich.edu/~mejn/netdata/)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>read_gml(path[, relabel])</code></td>
<td>Read graph in GML format from path.</td>
</tr>
<tr>
<td><code>write_gml(G, path)</code></td>
<td>Write the graph G in GML format to the file or file handle path.</td>
</tr>
<tr>
<td><code>parse_gml(lines[, relabel])</code></td>
<td>Parse GML graph from a string or iterable.</td>
</tr>
<tr>
<td><code>generate_gml(G)</code></td>
<td>Generate a single entry of the graph G in GML format.</td>
</tr>
</tbody>
</table>

#### 9.5.2 read_gml

**read_gml** *(path, relabel=False)*

Read graph in GML format from path.
Parameters  

- **path**: filename or filehandle
  
The filename or filehandle to read from.

- **relabel**: bool, optional
  
  If True use the GML node label attribute for node names otherwise use the node id.

Returns

- **G**: MultiGraph or MultiDiGraph

Raises

- **ImportError**
  
  If the pyparsing module is not available.

See Also:

- write_gml, parse_gml

Notes

Requires pyparsing: [http://pyparsing.wikispaces.com/](http://pyparsing.wikispaces.com/) The GML specification says that files should be ASCII encoded, with any extended ASCII characters (iso8859-1) appearing as HTML character entities.

References

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

Examples

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

### 9.5.3 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).

This implementation does not support all Python data types as GML data. Nodes, node attributes, edge attributes, and graph attributes must be either dictionaries or single stings or numbers. If they are not an attempt is made to represent them as strings. For example, a list as edge data G[1][2][‘somedata’]=[1,2,3], will be represented in the GML file as:
edge [  
  source 1  
  target 2  
  somedata "[1, 2, 3]"
]

Examples

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

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

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

### 9.5.4 parse_gml

`parse_gml(lines, relabel=True)`  
Parse GML graph from a string or iterable.

- **Parameters**  
  - `lines` : string or iterable  
    Data in GML format.
  - `relabel` : bool, optional  
    If True use the GML node label attribute for node names otherwise use the node id.

- **Returns**  
  - `G` : MultiGraph or MultiDiGraph

- **Raises**  
  - `ImportError`  
    If the pyparsing module is not available.

See Also:

- `write_gml`, `read_gml`

Notes

This stores nested GML attributes as dictionaries in the NetworkX graph, node, and edge attribute structures.

Requires pyparsing: [http://pyparsing.wikispaces.com/](http://pyparsing.wikispaces.com/)

References

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

### 9.5.5 generate_gml

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

- **Parameters**  
  - `G` : NetworkX graph
Returns lines: string

Lines in GML format.

Notes

This implementation does not support all Python data types as GML data. Nodes, node attributes, edge attributes, and graph attributes must be either dictionaries or single strings or numbers. If they are not an attempt is made to represent them as strings. For example, a list as edge data G[1][2][‘somedata’]=[1,2,3], will be represented in the GML file as:

```
edge [ source 1
target 2
somedata "[1, 2, 3]"
]
```

9.6 Pickle

Read and write NetworkX graphs as Python pickles.

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

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

9.6.1 Format

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

<table>
<thead>
<tr>
<th>read_gpickle(path)</th>
<th>Read graph object in Python pickle format.</th>
</tr>
</thead>
<tbody>
<tr>
<td>write_gpickle(G, path)</td>
<td>Write graph in Python pickle format.</td>
</tr>
</tbody>
</table>

9.6.2 read_gpickle

read_gpickle(path)

Read graph object in Python pickle format.

Pickles are a serialized byte stream of a Python object [R324]. 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

[R324]

Examples

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

9.6.3 write_gpickle

`write_gpickle(G, path)`
Write graph in Python pickle format.

Pickles are a serialized byte stream of a Python object [R325]. 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

[R325]

Examples

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

9.7 GraphML

Read and write graphs in GraphML format.

This implementation does not support mixed graphs (directed and undirected edges together), hyperedges, nested graphs, or ports.

“GraphML is a comprehensive and easy-to-use file format for graphs. It consists of a language core to describe the structural properties of a graph and a flexible extension mechanism to add application-specific data. Its main features include support of

- directed, undirected, and mixed graphs,
- hypergraphs,
- hierarchical graphs,
- graphical representations,
• references to external data,
• application-specific attribute data, and
• light-weight parsers.

Unlike many other file formats for graphs, GraphML does not use a custom syntax. Instead, it is based on XML and hence ideally suited as a common denominator for all kinds of services generating, archiving, or processing graphs.

http://graphml.graphdrawing.org/

9.7.1 Format

GraphML is an XML format. See http://graphml.graphdrawing.org/specification.html for the specification and http://graphml.graphdrawing.org/primer/graphml-primer.html for examples.

```
read_graphml(path[, node_type]) Read graph in GraphML format from path.
write_graphml(G, path[, encoding, prettyprint]) Write G in GraphML XML format to path
```

9.7.2 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.
- **node_type**: Python type (default: str)
  
  Convert node ids to this type

Returns

- **graph**: NetworkX graph
  
  If no parallel edges are found a Graph or DiGraph is returned. Otherwise a MultiGraph or MultiDiGraph is returned.

Notes

This implementation does not support mixed graphs (directed and undirected edges together), hypergraphs, nested graphs, or ports.

For multigraphs the GraphML edge “id” will be used as the edge key. If not specified then they “key” attribute will be used. If there is no “key” attribute a default NetworkX multigraph edge key will be provided.

Files with the yEd “yfiles” extension will can be read but the graphics information is discarded.

yEd compressed files (“file.graphmlz” extension) can be read by renaming the file to “file.graphml.gz”.

9.7.3 write_graphml

```
write_graphml(G, path[, encoding='utf-8', prettyprint=True]) Write G in GraphML XML format to path
```

Parameters

- **G**: graph
  
  A networkx graph
**path** : file or string

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

**encoding** : string (optional)

Encoding for text data.

**prettyprint** : bool (optional)

If True use line breaks and indenting in output XML.

**Notes**

This implementation does not support mixed graphs (directed and unidirected edges together) hyperedges, nested graphs, or ports.

**Examples**

```python
>>> G=nx.path_graph(4)
>>> nx.write_graphml(G, "test.graphml")
```

### 9.8 JSON

Generate and parse JSON serializable data for NetworkX graphs.

These formats are suitable for use with the d3.js examples http://d3js.org/

The three formats that you can generate with NetworkX are:

- node-link like in the d3.js example http://bl.ocks.org/mbostock/4062045
- tree like in the d3.js example http://bl.ocks.org/mbostock/4063550
- adjacency like in the d3.js example http://bost.ocks.org/mike/miserables/

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>node_link_data(G, attrs)</code></td>
<td>Return data in node-link format that is suitable for JSON serialization</td>
</tr>
<tr>
<td><code>node_link_graph(data[, directed, ...])</code></td>
<td>Return graph from node-link data format.</td>
</tr>
<tr>
<td><code>adjacency_data(G[, attrs])</code></td>
<td>Return data in adjacency format that is suitable for JSON serialization</td>
</tr>
<tr>
<td><code>adjacency_graph(data[, directed, ...])</code></td>
<td>Return graph from adjacency data format.</td>
</tr>
<tr>
<td><code>tree_data(G, root[, attrs])</code></td>
<td>Return data in tree format that is suitable for JSON serialization</td>
</tr>
<tr>
<td><code>tree_graph(data[, attrs])</code></td>
<td>Return graph from tree data format.</td>
</tr>
</tbody>
</table>

#### 9.8.1 node_link_data

**node_link_data** *(G, attrs=\{'source': 'source', 'target': 'target', 'key': 'key', 'id': 'id'\})*

Return data in node-link format that is suitable for JSON serialization and use in Javascript documents.

**Parameters**

- **G** : NetworkX graph
- **attrs** : dict

A dictionary that contains four keys 'id', 'source', 'target' and 'key'. The corresponding values provide the attribute names for storing NetworkX-internal graph data. The values should be unique. Default value: `dict(id='id', source='source',`
If some user-defined graph data use these attribute names as data keys, they may be silently dropped.

Returns data : dict
A dictionary with node-link formatted data.

Raises NetworkXError
If values in attrs are not unique.

See Also:
node_link_graph, adjacency_data, tree_data

Notes
Graph, node, and link attributes are stored in this format but keys for attributes must be strings if you want to serialize with JSON.
The default value of attrs will be changed in a future release of NetworkX.

Examples

```python
>>> from networkx.readwrite import json_graph
>>> G = nx.Graph([(1,2)])
>>> data = json_graph.node_link_data(G)

To serialize with json
```

```python
>>> import json
>>> s = json.dumps(data)
```

9.8.2 node_link_graph

node_link_graph(data, directed=False, multigraph=True, attrs={'source': 'source', 'target': 'target', 'key': 'key', 'id': 'id'})

Return graph from node-link data format.

Parameters data : dict
node-link formatted graph data

directed : bool
If True, and direction not specified in data, return a directed graph.

multigraph : bool
If True, and multigraph not specified in data, return a multigraph.

attrs : dict
A dictionary that contains four keys ‘id’, ‘source’, ‘target’ and ‘key’. The corresponding values provide the attribute names for storing NetworkX-internal graph data. Default value: dict(id='id', source='source', target='target', key='key').
Returns  $G$: NetworkX graph

A NetworkX graph object

See Also:

node_link_data, adjacency_data, tree_data

Notes

The default value of attrs will be changed in a future release of NetworkX.

Examples

```python
>>> from networkx.readwrite import json_graph
>>> G = nx.Graph([(1,2)])
>>> data = json_graph.node_link_data(G)
>>> H = json_graph.node_link_graph(data)
```

9.8.3 adjacency_data

**adjacency_data** $(G, attrs=\{\'id\': \'id\', \'key\': \'key\'}$)

Return data in adjacency format that is suitable for JSON serialization and use in Javascript documents.

**Parameters**  

$G$: NetworkX graph

**attrs**: dict

A dictionary that contains two keys `id` and `key`. The corresponding values provide the attribute names for storing NetworkX-internal graph data. The values should be unique. Default value: `dict(id=\'id\', key=\'key\')`.

If some user-defined graph data use these attribute names as data keys, they may be silently dropped.

**Returns**  

$data$: dict

A dictionary with adjacency formatted data.

**Raises**  

NetworkXError

If values in attrs are not unique.

See Also:

adjacency_graph, node_link_data, tree_data

Notes

Graph, node, and link attributes will be written when using this format but attribute keys must be strings if you want to serialize the resulting data with JSON.

The default value of attrs will be changed in a future release of NetworkX.
Examples

```python
>>> from networkx.readwrite import json_graph
>>> G = nx.Graph([(1, 2)])
>>> data = json_graph.adjacency_data(G)

To serialize with json

```python
>>> import json
>>> s = json.dumps(data)
```

9.8.4 adjacency_graph

adjacency_graph (data, directed=False, multigraph=True, attrs={'id': 'id', 'key': 'key'})

Return graph from adjacency data format.

Parameters:
- **data**: dict
  Adjacency list formatted graph data

Returns:
- **G**: NetworkX graph
  A NetworkX graph object

- **directed**: bool
  If True, and direction not specified in data, return a directed graph.

- **multigraph**: bool
  If True, and multigraph not specified in data, return a multigraph.

- **attrs**: dict
  A dictionary that contains two keys ‘id’ and ‘key’. The corresponding values provide
  the attribute names for storing NetworkX-internal graph data. The values should be
  unique. Default value: dict(id='id', key='key').

See Also:
adjacency_graph, node_link_data, tree_data

Notes

The default value of attrs will be changed in a future release of NetworkX.

Examples

```python
>>> from networkx.readwrite import json_graph
>>> G = nx.Graph([(1, 2)])
>>> data = json_graph.adjacency_data(G)
>>> H = json_graph.adjacency_graph(data)
```
9.8.5 tree_data

tree_data(G, root, attrs=\{‘children’: ‘children’, ‘id’: ‘id’\})

Return data in tree format that is suitable for JSON serialization and use in Javascript documents.

**Parameters**

- **G** : NetworkX graph
  - G must be an oriented tree
- **root** : node
  - The root of the tree
- **attrs** : dict
  - A dictionary that contains two keys ‘id’ and ‘children’. The corresponding values provide the attribute names for storing NetworkX-internal graph data. The values should be unique. Default value: dict(id=’id’, children=’children’).
  - If some user-defined graph data use these attribute names as data keys, they may be silently dropped.

**Returns**

- **data** : dict
  - A dictionary with node-link formatted data.

**Raises**

- NetworkXError
  - If values in attrs are not unique.

**See Also:**

tree_graph, node_link_data, node_link_data

**Notes**

- Node attributes are stored in this format but keys for attributes must be strings if you want to serialize with JSON.
- Graph and edge attributes are not stored.
- The default value of attrs will be changed in a future release of NetworkX.

**Examples**

```python
>>> from networkx.readwrite import json_graph
>>> G = nx.DiGraph([(1, 2)])
>>> data = json_graph.tree_data(G, root=1)

To serialize with json
```

```python
>>> import json
>>> s = json.dumps(data)
```

9.8.6 tree_graph

tree_graph(data, attrs=\{‘children’: ‘children’, ‘id’: ‘id’\})

Return graph from tree data format.

**Parameters**

- **data** : dict
Tree formatted graph data

**Returns**  
G : NetworkX DiGraph

**attrs** : dict

A dictionary that contains two keys ‘id’ and ‘children’. The corresponding values provide the attribute names for storing NetworkX-internal graph data. The values should be unique. Default value: dict(id='id', children='children').

**See Also:**

tree_graph, node_link_data, adjacency_data

**Notes**

The default value of attrs will be changed in a future release of NetworkX.

**Examples**

```python
>>> from networkx.readwrite import json_graph

>>> G = nx.DiGraph([(1,2)])

>>> data = json_graph.tree_data(G, root=1)

>>> H = json_graph.tree_graph(data)
```

### 9.9 LEDA

Read graphs in LEDA format.

LEDA is a C++ class library for efficient data types and algorithms.

#### 9.9.1 Format

See [http://www.algorithmic-solutions.info/leda_guide/graphs/leda_native_graph_fileformat.html](http://www.algorithmic-solutions.info/leda_guide/graphs/leda_native_graph_fileformat.html)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_leda(path[, encoding])</td>
<td>Read graph in LEDA format from path.</td>
</tr>
<tr>
<td>parse_leda(lines)</td>
<td>Read graph in LEDA format from string or iterable.</td>
</tr>
</tbody>
</table>

#### 9.9.2 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**

[R327]
Examples

G=nx.read_leda('file.leda')

9.9.3 parse_lleda

parse_lleda(lines)
Read graph in LEDA format from string or iterable.

Parameters
lines : string or iterable
Data in LEDA format.

Returns
G : NetworkX graph

References

[R326]

Examples

G=nx.parse_lleda(string)

9.10 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.

9.10.1 Format

http://pyyaml.org/wiki/PyYAML

| read_yaml(path) | Read graph in YAML format from path. |
| write_yaml(G, path[, encoding]) | Write graph G in YAML format to path. |

9.10.2 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 [R330].

Parameters
path : file or string
File or filename to read. Filenames ending in .gz or .bz2 will be uncompressed.

Returns
G : NetworkX graph
References

[R330]

Examples

```python
>>> G=nx.path_graph(4)
>>> nx.write_yaml(G,'test.yaml')
>>> G=nx.read_yaml('test.yaml')
```

9.10.3 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 [R331].

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

[R331]

Examples

```python
>>> G=nx.path_graph(4)
>>> nx.write_yaml(G,'test.yaml')
```

9.11 SparseGraph6

9.11.1 Graph6

Graph6

Read and write graphs in graph6 format.
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.”


<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parse_graph6(string)</td>
<td>Read a simple undirected graph in graph6 format from string.</td>
</tr>
<tr>
<td>read_graph6(path)</td>
<td>Read simple undirected graphs in graph6 format from path.</td>
</tr>
<tr>
<td>generate_graph6(G[, nodes, header])</td>
<td>Generate graph6 format string from a simple undirected graph.</td>
</tr>
<tr>
<td>write_graph6(G, path[, nodes, header])</td>
<td>Write a simple undirected graph to path in graph6 format.</td>
</tr>
</tbody>
</table>

**parse_graph6**

parse_graph6(string)

Read a simple undirected graph in graph6 format from string.

**Parameters**

string : string

Data in graph6 format

**Returns**

G : Graph

**Raises**

NetworkXError

If the string is unable to be parsed in graph6 format

**See Also:**

generate_graph6, read_graph6, write_graph6

**References**


**Examples**

```python
>>> G = nx.parse_graph6('A_')
>>> sorted(G.edges())
[(0, 1)]
```

**read_graph6**

read_graph6(path)

Read simple undirected graphs in graph6 format from path.

**Parameters**

path : file or string

File or filename to write.

**Returns**

G : Graph or list of Graphs

If the file contains multiple lines then a list of graphs is returned

**Raises**

NetworkXError

If the string is unable to be parsed in graph6 format
See Also:
generate_graph6, parse_graph6, write_graph6

References


Examples

```python
>>> nx.write_graph6(nx.Graph([(0,1)]), 'test.g6')
>>> G = nx.read_graph6('test.g6')
>>> sorted(G.edges())
[(0, 1)]
```

**generate_graph6**

`generate_graph6(G, nodes=None, header=True)`
Generate graph6 format string from a simple undirected graph.

Parameters

- **G**: Graph (undirected)

  nodes: list or iterable
    
    Nodes are labeled 0...n-1 in the order provided. If None the ordering given by G.nodes() is used.

  header: bool
    
    If True add ‘>>graph6<<’ string to head of data

Returns

- **s**: string

  String in graph6 format

Raises

- **NetworkXError**

  If the graph is directed or has parallel edges

See Also:

read_graph6, parse_graph6, write_graph6

Notes

The format does not support edge or node labels, parallel edges or self loops. If self loops are present they are silently ignored.

References

Examples

```python
>>> G = nx.Graph([(0, 1)])
>>> nx.generate_graph6(G)
'>>graph6<<A_'
```

write_graph6

```python
write_graph6(G, path, nodes=None, header=True)
```
Write a simple undirected graph to path in graph6 format.

**Parameters**
- **G**: Graph (undirected)
- **path**: file or string
  - File or filename to write.
- **nodes**: list or iterable
  - Nodes are labeled 0...n-1 in the order provided. If None the ordering given by G.nodes() is used.
- **header**: bool
  - If True add ‘>>graph6<<’ string to head of data

**Raises**
- NetworkXError
  - If the graph is directed or has parallel edges

**See Also**
- generate_graph6, parse_graph6, read_graph6

**Notes**
The format does not support edge or node labels, parallel edges or self loops. If self loops are present they are silently ignored.

**References**

**Examples**

```python
>>> G = nx.Graph([(0, 1)])
>>> nx.write_graph6(G, 'test.g6')
```

9.11.2 Sparse6

Sparse6
Read and write graphs in sparse6 format.
NetworkX Reference, Release 1.9

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.”


<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>parse_sparse6</code></td>
<td>Read an undirected graph in sparse6 format from string.</td>
</tr>
<tr>
<td><code>read_sparse6</code></td>
<td>Read an undirected graph in sparse6 format from path.</td>
</tr>
<tr>
<td><code>generate_sparse6</code></td>
<td>Generate sparse6 format string from an undirected graph.</td>
</tr>
<tr>
<td><code>write_sparse6</code></td>
<td>Write graph G to given path in sparse6 format.</td>
</tr>
</tbody>
</table>

`parse_sparse6`

`parse_sparse6(string)`

Read an undirected graph in sparse6 format from string.

**Parameters**

- `string`: string
  - Data in sparse6 format

**Returns**

- `G`: Graph

**Raises**

- `NetworkXError`: If the string is unable to be parsed in sparse6 format

**See Also:**

`generate_sparse6`, `read_sparse6`, `write_sparse6`

**References**

Sparse6 specification: http://cs.anu.edu.au/~bdm/data/formats.txt

**Examples**

```python
>>> G = nx.parse_sparse6(':A_')
>>> sorted(G.edges())
[(0, 1), (0, 1), (0, 1)]
```

`read_sparse6`

`read_sparse6(path)`

Read an undirected graph in sparse6 format from path.

**Parameters**

- `path`: file or string
  - File or filename to write.

**Returns**

- `G`: Graph/Multigraph or list of Graphs/MultiGraphs
  - If the file contains multiple lines then a list of graphs is returned

**Raises**

- `NetworkXError`: If the string is unable to be parsed in sparse6 format
See Also:

`generate_sparse6, read_sparse6, parse_sparse6`

References

Sparse6 specification: http://cs.anu.edu.au/~bdm/data/formats.txt

Examples

```python
>>> nx.write_sparse6(nx.Graph([(0,1),(0,1),(0,1)]), 'test.s6')
>>> G = nx.read_sparse6('test.s6')
>>> sorted(G.edges())
[(0, 1)]
```

`generate_sparse6`

`generate_sparse6(G, nodes=None, header=True)`

Generate sparse6 format string from an undirected graph.

Parameters

- `G`: Graph (undirected)
- `nodes`: list or iterable
  - Nodes are labeled 0...n-1 in the order provided. If None the ordering given by G.nodes() is used.
- `header`: bool
  - If True add `>>sparse6<<` string to head of data

Returns

- `s`: string
  - String in sparse6 format

Raises

- `NetworkXError`
  - If the graph is directed

See Also:

`read_sparse6, parse_sparse6, write_sparse6`

Notes

The format does not support edge or node labels. References ———- Sparse6 specification: http://cs.anu.edu.au/~bdm/data/formats.txt for details.

Examples

```python
>>> G = nx.MultiGraph([(0, 1), (0, 1), (0, 1)])
>>> nx.generate_sparse6(G)
'>>sparse6<<:A_`
```
write_sparse6

write_sparse6(G, path, nodes=None, header=True)
Write graph G to given path in sparse6 format. Parameters ———- G : Graph (undirected)
path [file or string] File or filename to write
nodes: list or iterable Nodes are labeled 0...n-1 in the order provided. If None the ordering given by G.nodes() is used.
header: bool If True add ‘>>sparse6<<’ string to head of data

Raises NetworkXError
If the graph is directed

See Also:
read_sparse6, parse_sparse6, generate_sparse6

Notes
The format does not support edge or node labels.

References

Examples

>>> G = nx.Graph([(0, 1), (0, 1), (0, 1)])
>>> nx.write_sparse6(G, 'test.s6')

9.12 Pajek

Read graphs in Pajek format.
This implementation handles directed and undirected graphs including those with self loops and parallel edges.

9.12.1 Format


<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_pajek(path[, encoding])</td>
<td>Read graph in Pajek format from path.</td>
</tr>
<tr>
<td>write_pajek(G, path[, encoding])</td>
<td>Write graph in Pajek format to path.</td>
</tr>
<tr>
<td>parse_pajek(lines)</td>
<td>Parse Pajek format graph from string or iterable.</td>
</tr>
</tbody>
</table>
9.12.2 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**

**Examples**

```python
>>> 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

```python
>>> G1=nx.Graph(G)
```

9.12.3 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**

**Examples**

```python
>>> G=nx.path_graph(4)
>>> nx.write_pajek(G, "test.net")
```

9.12.4 parse_pajek

**parse_pajek** *(lines)*
Parse Pajek format graph from string or iterable.

**Parameters**
- **lines**: string or iterable

9.12. Pajek
Data in Pajek format.

**Returns**  
G : NetworkX graph

**See Also:**
read_pajek

## 9.13 GIS Shapefile

Generates a networkx.DiGraph from point and line shapefiles.

“The Esri Shapefile or simply a shapefile is a popular geospatial vector data format for geographic information systems software. It is developed and regulated by Esri as a (mostly) open specification for data interoperability among Esri and other software products.” See [http://en.wikipedia.org/wiki/Shapefile](http://en.wikipedia.org/wiki/Shapefile) for additional information.

### 9.13.1 read_shp

**read_shp** *(path)*
Generates a networkx.DiGraph from shapefiles. Point geometries are translated into nodes, lines into edges. Coordinate tuples are used as keys. Attributes are preserved, line geometries are simplified into start and end coordinates. Accepts a single shapefile or directory of many shapefiles.

“The Esri Shapefile or simply a shapefile is a popular geospatial vector data format for geographic information systems software [R328].”

**Parameters**  
**path** : file or string  
File, directory, or filename to read.

**Returns**  
G : NetworkX graph

**References**

[R328]

**Examples**

```python
>>> G=nx.read_shp('test.shp')
```

### 9.13.2 write_shp

**write_shp** *(G, outdir)*  
Writes a networkx.DiGraph to two shapefiles, edges and nodes. Nodes and edges are expected to have a Well Known Binary (Wkb) or Well Known Text (Wkt) key in order to generate geometries. Also acceptable are nodes with a numeric tuple key (x,y).

“The Esri Shapefile or simply a shapefile is a popular geospatial vector data format for geographic information systems software [R329].”

**References**

[R329]
Parameters  outdir : directory path
   Output directory for the two shapefiles.

Returns  None

References

[R329]

Examples

nx.write_shp(digraph, ‘/shapefiles’) # doctest +SKIP
10.1 Matplotlib

Draw networks with matplotlib.

10.1.1 See Also

matplotlib: http://matplotlib.sourceforge.net/
pygraphviz: http://networkx.lanl.gov/pygraphviz/

draw(G[, pos, ax, hold])
Draw the graph G with Matplotlib.

draw_networkx(G[, pos, with_labels])
Draw the graph G using Matplotlib.

draw_networkx_nodes(G[, nodelist, ...])
Draw the nodes of the graph G.

draw_networkx_edges(G[, edgelist, ...])
Draw the edges of the graph G.

draw_networkx_labels(G[, labels, ...])
Draw node labels on the graph G.

draw_networkx_edge_labels(G[, ...])
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[, prog])
Draw networkx graph with graphviz layout.

10.1.2 draw

draw(G, pos=None, ax=None, hold=None, **kwds)
Draw the graph G with Matplotlib.

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.

With pyplot use

>>> 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 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.

**with_labels** : bool, optional (default=True)
Set to True to draw labels on the nodes.

**ax** : Matplotlib Axes object, optional
Draw the graph in the specified Matplotlib axes.

**nodelist** : list, optional (default G.nodes())
Draw only specified nodes

**edgelist** : list, optional (default=G.edges())
Draw only specified edges

**node_size** : scalar or array, optional (default=300)
Size of nodes. If an array is specified it must be the same length as nodelist.

**node_color** : color string, or array of floats, (default='r')
Node color. Can be a single color format string, 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, optional (default='o')
The shape of the node. Specification is as matplotlib.scatter marker, one of 'so^>v<dph8'.

**alpha** : float, optional (default=1.0)
The node transparency

**cmap** : Matplotlib colormap, optional (default=None)
Colormap for mapping intensities of nodes

**vmin,vmax** : float, optional (default=None)
Minimum and maximum for node colormap scaling

**linewidths** : [None | scalar | sequence]
Line width of symbol border (default =1.0)

**width** : float, optional (default=1.0)
Line width of edges

**edge_color** : color string, or array of floats (default='r')
Edge color. Can be a single color format string, 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, optional (default=None)
Colormap for mapping intensities of edges

**edge_vmin,edge_vmax** : floats, optional (default=None)
Minimum and maximum for edge colormap scaling
style : string, optional (default='solid')
    Edge line style (solid|dashed|dotted,dashdot)
labels : dictionary, optional (default=None)
    Node labels in a dictionary keyed by node of text labels
font_size : int, optional (default=12)
    Font size for text labels
font_color : string, optional (default='k' black)
    Font color string
font_weight : string, optional (default='normal')
    Font weight
font_family : string, optional (default='sans-serif')
    Font family
label : string, optional
    Label for graph legend

See Also:
draw, draw_networkx_nodes, draw_networkx_edges, draw_networkx_labels,
draw_networkx_edge_labels

Examples

>>> G=nx.dodecahedral_graph()
>>> nx.draw(G)
>>> nx.draw(G,pos=nx.spring_layout(G)) # use spring layout

>>> import matplotlib.pyplot as plt
>>> limits=plt.axis('off') # turn of axis

Also see the NetworkX drawing examples at http://networkx.lanl.gov/gallery.html

10.1.4 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, label=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. Positions should be sequences
        of length 2.
    ax : Matplotlib Axes object, optional
Draw the graph in the specified Matplotlib axes.

**nodelist** : list, optional

Draw only specified nodes (default G.nodes())

**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)

**linewidths** : [None | scalar | sequence]

   Line width of symbol border (default =1.0)

**label** : [None | string]

Label for legend

**Returns**

    matplotlib.collections.PathCollection

PathCollection of the nodes.

See Also:

draw, draw_networkx, draw_networkx_edges, draw_networkx_labels, draw_networkx_edge_labels

Examples

```python
>>> 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 draw_networkx_edges

\textbf{draw_networkx_edges} \((G, \text{pos, edgelist=None, width=1.0, edge_color='k', style='solid', alpha=None, edge_cmap=None, edge_vmin=None, edge_vmax=None, ax=None, arrows=True, label=None, **kwds})\)

Draw the edges of the graph \(G\).

This draws only the edges of the graph \(G\).

**Parameters**

\(G\) : graph

A networkx graph

\(\text{pos}\) : dictionary

A dictionary with nodes as keys and positions as values. Positions should be sequences of length 2.

\(\text{edgelist}\) : collection of edge tuples

Draw only specified edges (default=G.edges())

\(\text{width}\) : float

Line width of edges (default=1.0)

\(\text{edge_color}\) : 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.

\(\text{style}\) : string

Edge line style (default='solid') (solid|dashed|dotted,dashdot)

\(\text{alpha}\) : float

The edge transparency (default=1.0)

\(\text{edge_cmap}\) : Matplotlib colormap

Colormap for mapping intensities of edges (default=None)

\(\text{edge_vmin,edge_vmax}\) : floats

Minimum and maximum for edge colormap scaling (default=None)

\(\text{ax}\) : Matplotlib Axes object, optional

Draw the graph in the specified Matplotlib axes.

\(\text{arrows}\) : bool, optional (default=True)

For directed graphs, if True draw arrowheads.

\(\text{label}\) : [None| string]

Label for legend

**Returns**

\text{matplotlib.collection.LineCollection}

\text{LineCollection} of the edges

\textbf{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

```python
>>> 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 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
  - A dictionary with nodes as keys and positions as values. Positions should be sequences of length 2.
- **labels**: dictionary, optional (default=None)
  - Node labels in a dictionary keyed by node of text labels
- **font_size**: int
  - Font size for text labels (default=12)
- **font_color**: string
  - Font color string (default='k' black)
- **font_family**: string
  - Font family (default='sans-serif')
- **font_weight**: string
  - Font weight (default='normal')
- **alpha**: float
  - The text transparency (default=1.0)
- **ax**: Matplotlib Axes object, optional
  - Draw the graph in the specified Matplotlib axes.

Returns

- dict
  - dict of labels keyed on the nodes

See Also:

draw, draw_networkx, draw_networkx_nodes, draw_networkx_edges, draw_networkx_edge_labels

10.1. Matplotlib
Examples

```python
>>> 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 draw_networkx_edge_labels

`draw_networkx_edge_labels(G, pos, edge_labels=None, label_pos=0.5, font_size=10, font_color='k', font_family='sans-serif', font_weight='normal', alpha=1.0, bbox=None, ax=None, rotate=True, **kwds)`

Draw edge labels.

**Parameters**

- **G : graph**
  A networkx graph

- **pos : dictionary**
  A dictionary with nodes as keys and positions as values. Positions should be sequences of length 2.

- **ax : Matplotlib Axes object, optional**
  Draw the graph in the specified Matplotlib axes.

- **alpha : float**
  The text transparency (default=1.0)

- **edge_labels : dictionary**
  Edge labels in a dictionary keyed by edge two-tuple of text labels (default=None). Only labels for the keys in the dictionary are drawn.

- **label_pos : float**
  Position of edge label along edge (0=head, 0.5=center, 1=tail)

- **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)

**Returns**

- **dict**
dict of labels keyed on the edges

See Also:
draw, draw_networkx, draw_networkx_nodes, draw_networkx_edges, draw_networkx_labels

Examples

```python
>>> 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 draw_circular

draw_circular(G, **kwargs)
   Draw the graph G with a circular layout.

   Parameters
   G : graph
       A networkx graph

   **kwargs : optional keywords
       See networkx.draw_networkx() for a description of optional keywords, with the excep-
       tion of the pos parameter which is not used by this function.

10.1.9 draw_random

draw_random(G, **kwargs)
   Draw the graph G with a random layout.

   Parameters
   G : graph
       A networkx graph

   **kwargs : optional keywords
       See networkx.draw_networkx() for a description of optional keywords, with the excep-
       tion of the pos parameter which is not used by this function.

10.1.10 draw_spectral

draw_spectral(G, **kwargs)
   Draw the graph G with a spectral layout.

   Parameters
   G : graph
       A networkx graph

   **kwargs : optional keywords
       See networkx.draw_networkx() for a description of optional keywords, with the excep-
       tion of the pos parameter which is not used by this function.
10.1.11 draw_spring

draw_spring(G, **kwargs)
Draw the graph G with a spring layout.

Parameters  
G : graph
A networkx graph

**kwargs : optional keywords
See networkx.draw_networkx() for a description of optional keywords, with the exception of the pos parameter which is not used by this function.

10.1.12 draw_shell

draw_shell(G, **kwargs)
Draw networkx graph with shell layout.

Parameters  
G : graph
A networkx graph

**kwargs : optional keywords
See networkx.draw_networkx() for a description of optional keywords, with the exception of the pos parameter which is not used by this function.

10.1.13 draw_graphviz

draw_graphviz(G, prog='neato', **kwargs)
Draw networkx graph with graphviz layout.

Parameters  
G : graph
A networkx graph

prog : string, optional
Name of Graphviz layout program

**kwargs : optional keywords
See networkx.draw_networkx() for a description of optional keywords.

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>from_agraph(A[, create_using])</td>
<td>Return a NetworkX Graph or DiGraph from a PyGraphviz graph.</td>
</tr>
<tr>
<td>to_agraph(N)</td>
<td>Return a pygraphviz graph from a NetworkX graph N.</td>
</tr>
<tr>
<td>write_dot(G, path)</td>
<td>Write NetworkX graph G to Graphviz dot format on path.</td>
</tr>
<tr>
<td>read_dot(path)</td>
<td>Return a NetworkX graph from a dot file on path.</td>
</tr>
<tr>
<td>graphviz_layout(G[, prog, root, args])</td>
<td>Create node positions for G using Graphviz.</td>
</tr>
<tr>
<td>pygraphviz_layout(G[, prog, root, args])</td>
<td>Create node positions for G using Graphviz.</td>
</tr>
</tbody>
</table>

10.2.3 from_agraph

```python
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**

```python
>>> K5=nx.complete_graph(5)
>>> A=nx.to_agraph(K5)
>>> G=nx.from_agraph(A)
>>> G=nx.from_agraph(A)
```

10.2.4 to_agraph

```python
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

```python
>>> K5=nx.complete_graph(5)
>>> A=nx.to_agraph(K5)
```

10.2.5 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 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 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

```python
>>> G=nx.petersen_graph()
>>> pos=nx.graphviz_layout(G)
>>> pos=nx.graphviz_layout(G,prog='dot')
```

10.2.8 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 of x,y, positions keyed by node.

Examples

```python
>>> 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://code.google.com/p/pydot/  

```
from_pydot(P)  
Return a NetworkX graph from a Pydot graph.
```

Continued on next page
Table 10.3 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>to_pydot(N[, strict])</td>
<td>Return a pydot graph from a NetworkX graph N.</td>
</tr>
<tr>
<td>write_dot(G, path)</td>
<td>Write NetworkX graph G to Graphviz dot format on path.</td>
</tr>
<tr>
<td>read_dot(path)</td>
<td>Return a NetworkX MultiGraph or MultiDiGraph from a dot file on path.</td>
</tr>
<tr>
<td>graphviz_layout(G[, prog, root])</td>
<td>Create node positions using Pydot and Graphviz.</td>
</tr>
<tr>
<td>pydot_layout(G[, prog, root])</td>
<td>Create node positions using Pydot and Graphviz.</td>
</tr>
</tbody>
</table>

### 10.3.2 from_pydot

`from_pydot(P)`

Return a NetworkX graph from a Pydot graph.

**Parameters**
- `P` : Pydot graph
  - A graph created with Pydot

**Returns**
- `G` : NetworkX multigraph
  - A MultiGraph or MultiDiGraph.

**Examples**

```python
>>> K5=nx.complete_graph(5)
>>> A=nx.to_pydot(K5)
>>> G=nx.from_pydot(A)  # return MultiGraph
>>> G=nx.Graph(nx.from_pydot(A))  # make a Graph instead of MultiGraph
```

### 10.3.3 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**

```python
>>> K5=nx.complete_graph(5)
>>> P=nx.to_pydot(K5)
```

### 10.3.4 write_dot

`write_dot(G, path)`

Write NetworkX graph G to Graphviz dot format on path.

- Path can be a string or a file handle.
10.3.5 read_dot

read_dot(path)
Return a NetworkX MultiGraph or MultiDiGraph from a dot file on path.

Parameters  path : filename or file handle

Returns  G : NetworkX multigraph
A MultiGraph or MultiDiGraph.

Notes
Use G=nx.Graph(nx.read_dot(path)) to return a Graph instead of a MultiGraph.

10.3.6 graphviz_layout

graphviz_layout(G, prog='neato', root=None, **kwds)
Create node positions using Pydot and Graphviz.
Returns a dictionary of positions keyed by node.

Notes
This is a wrapper for pydot_layout.

Examples

```python
>>> G=nx.complete_graph(4)
>>> pos=nx.graphviz_layout(G)
>>> pos=nx.graphviz_layout(G,prog='dot')
```

10.3.7 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

```python
>>> 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, k, pos, fixed, ...])  Position nodes using Fruchterman-Reingold force-directed algorithm.

spectral_layout(G[, dim, weight, scale])  Position nodes using the eigenvectors of the graph Laplacian.

10.4.1 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 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

```python
>>> G = nx.lollipop_graph(4, 3)
>>> pos = nx.random_layout(G)
```

10.4.3 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**

```python
>>> G=nx.path_graph(4)
>>> shells=[[0],[1,2,3]]
>>> pos=nx.shell_layout(G,shells)
```

10.4.4 spring_layout

**spring_layout** *(G, dim=2, k=None, pos=None, fixed=None, iterations=50, weight='weight', scale=1.0)*
Position nodes using Fruchterman-Reingold force-directed algorithm.

**Parameters**
- **G**: NetworkX graph
- **dim**: int
  - Dimension of layout
- **k**: float (default=None)
  - Optimal distance between nodes. If None the distance is set to 1/sqrt(n) where n is the number of nodes. Increase this value to move nodes farther apart.
- **pos**: dict or None optional (default=None)
  - Initial positions for nodes as a dictionary with node as keys and values as a list or tuple. If None, then use random initial positions.
**fixed**: list or None optional (default=None)
Nodes to keep fixed at initial position.

**iterations**: int optional (default=50)
Number of iterations of spring-force relaxation

**weight**: string or None optional (default='weight')
The edge attribute that holds the numerical value used for the edge weight. If None, then all edge weights are 1.

**scale**: float (default=1.0)
Scale factor for positions. The nodes are positioned in a box of size [0, scale] x [0, scale].

**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 spectral_layout

**spectral_layout** *(G, dim=2, weight='weight', scale=1)*
Position nodes using the eigenvectors of the graph Laplacian.

**Parameters**

**G**: NetworkX graph

**dim**: int
Dimension of layout

**weight**: string or None optional (default='weight')
The edge attribute that holds the numerical value used for the edge weight. If None, then all edge weights are 1.

**scale**: float
Scale factor for positions

**Returns**
dict:
A dictionary of positions keyed by node

**Notes**
Direct graphs will be considered as undirected graphs when positioning the nodes.
For larger graphs (>500 nodes) this will use the SciPy sparse eigenvalue solver (ARPACK).
Examples

```python
>>> G=nx.path_graph(4)
>>> pos=nx.spectral_layout(G)
```
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

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.
12.1 Helper Functions

Miscellaneous Helpers for NetworkX.

These are not imported into the base networkx namespace but can be accessed, for example, as

```python
>>> import networkx

>>> networkx.utils.is_string_like('spam')
True
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>is_string_like(obj)</code></td>
<td>Check if obj is string.</td>
</tr>
<tr>
<td><code>flatten(obj[, result])</code></td>
<td>Return flattened version of (possibly nested) iterable object.</td>
</tr>
<tr>
<td><code>iterable(obj)</code></td>
<td>Return True if obj is iterable with a well-defined len().</td>
</tr>
<tr>
<td><code>is_list_of_ints(intlist)</code></td>
<td>Return True if list is a list of ints.</td>
</tr>
<tr>
<td><code>make_str(x)</code></td>
<td>Return the string representation of x.</td>
</tr>
<tr>
<td><code>cumulative_sum(numbers)</code></td>
<td>Yield cumulative sum of numbers.</td>
</tr>
<tr>
<td><code>generate_unique_node()</code></td>
<td>Generate a unique node label.</td>
</tr>
<tr>
<td><code>default_opener(filename)</code></td>
<td>Opens <code>filename</code> using system’s default program.</td>
</tr>
</tbody>
</table>

12.1.1 `is_string_like`

`is_string_like(obj)`

Check if `obj` is string.

12.1.2 `flatten`

`flatten(obj[, result])`

Return flattened version of (possibly nested) iterable object.

12.1.3 `iterable`

`iterable(obj)`

Return True if `obj` is iterable with a well-defined len().
12.1.4 is_list_of_ints

is_list_of_ints(intlist)
Return True if list is a list of ints.

12.1.5 make_str

make_str(x)
Return the string representation of t.

12.1.6 cumulative_sum

cumulative_sum(numbers)
Yield cumulative sum of numbers.

>>> import networkx.utils as utils
>>> list(utils.cumulative_sum([1,2,3,4]))
[1, 3, 6, 10]

12.1.7 generate_unique_node

generate_unique_node()
Generate a unique node label.

12.1.8 default_opener

default_opener(filename)
Opens filename using system’s default program.
Parameters filename : str
The path of the file to be opened.

12.2 Data Structures and Algorithms

Union-find data structure.

- UnionFind.union(*objects) Find the sets containing the objects and merge them all.

12.2.1 union

UnionFind.union(*objects)
Find the sets containing the objects and merge them all.

12.3 Random Sequence Generators

Utilities for generating random numbers, random sequences, and random selections.
### 12.3.1 create_degree_sequence

**create_degree_sequence** \((n, \text{sfunction}, \text{max_tries})\)

Return sample sequence of length \(n\) from a Pareto distribution.

### 12.3.2 pareto_sequence

**pareto_sequence** \((n, \text{exponent}=1.0)\)

Return sample sequence of length \(n\) from a Pareto distribution.

### 12.3.3 powerlaw_sequence

**powerlaw_sequence** \((n, \text{exponent}=2.0)\)

Return sample sequence of length \(n\) from a power law distribution.

### 12.3.4 uniform_sequence

**uniform_sequence** \((n)\)

Return sample sequence of length \(n\) from a uniform distribution.

### 12.3.5 cumulative_distribution

**cumulative_distribution** \((\text{distribution})\)

Return normalized cumulative distribution from discrete distribution.

### 12.3.6 discrete_sequence

**discrete_sequence** \((n, \text{distribution}=\text{None}, \text{cdistribution}=\text{None})\)

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
ddistribution = normalized discrete cumulative distribution
12.3.7 zipf_sequence

`zipf_sequence(n, alpha=2.0, xmin=1)`
Return a sample sequence of length n from a Zipf distribution with exponent parameter alpha and minimum value xmin.

See Also:
`zipf_rv`

12.3.8 zipf_rv

`zipf_rv(alpha, xmin=1, seed=None)`
Return a random value chosen from the Zipf distribution.

The return value is an integer drawn from the probability distribution ::math:

\[ p(x) = \frac{x^{-\alpha}}{\zeta(\alpha,x_{min})}, \]

where \( \zeta(\alpha,x_{min}) \) is the Hurwitz zeta function.

Parameters  
- `alpha`: float
  Exponent value of the distribution
- `xmin`: int
  Minimum value
- `seed`: int
  Seed value for random number generator

Returns  
- `x`: int
  Random value from Zipf distribution

Raises  
- `ValueError`: 
  If xmin < 1 or If alpha <= 1

Notes

The rejection algorithm generates random values for a the power-law distribution in uniformly bounded expected time dependent on parameters. See [1] for details on its operation.

References


Examples

```python
>>> nx.zipf_rv(alpha=2, xmin=3, seed=42)
```
12.3.9 random_weighted_sample

random_weighted_sample(mapping, k)
Return k items without replacement from a weighted sample.
The input is a dictionary of items with weights as values.

12.3.10 weighted_choice

weighted_choice(mapping)
Return a single element from a weighted sample.
The input is a dictionary of items with weights as values.

12.4 Decorators

open_file(path_arg[, mode]) Decorator to ensure clean opening and closing of files.

12.4.1 open_file

open_file(path_arg, mode='r')
Decorator to ensure clean opening and closing of files.

Parameters
path_arg : int
Location of the path argument in args. Even if the argument is a named positional argument (with a default value), you must specify its index as a positional argument.

mode : str
String for opening mode.

Returns _open_file : function
Function which cleanly executes the io.

Examples

Decorate functions like this:

@open_file(0, ’r’)
def read_function(pathname):
   pass

@open_file(1, ’w’)
def write_function(G, pathname):
   pass

@open_file(1, ’w’)
def write_function(G, pathname=’graph.dot’)
   pass

@open_file(’path’, ’w+’)
def another_function(arg, **kwargs):
   pass
12.5 Cuthill-McKee Ordering

Cuthill-McKee ordering of graph nodes to produce sparse matrices

**cuthill_mckee_ordering**

Generate an ordering (permutation) of the graph nodes to make a sparse matrix.

Uses the Cuthill-McKee heuristic (based on breadth-first search) [R332].

**Parameters**

- **G**: graph
  
  A NetworkX graph

- **heuristic**: function, optional
  
  Function to choose starting node for RCM algorithm. If None a node from a psuedo-
  peripheral pair is used. A user-defined function can be supplied that takes a graph object
  and returns a single node.

**Returns**

- **nodes**: generator
  
  Generator of nodes in Cuthill-McKee ordering.

**See Also**

reverse_cuthill_mckee_ordering

**Notes**

The optimal solution the the bandwidth reduction is NP-complete [R333].

**References**

[R332], [R333]

**Examples**

```python
>>> from networkx.utils import cuthill_mckee_ordering
>>> G = nx.path_graph(4)
>>> rcm = list(cuthill_mckee_ordering(G))
>>> A = nx.adjacency_matrix(G, nodelist=rcm)
```

Smallest degree node as heuristic function:
>>> def smallest_degree(G):
...    node, deg = sorted(G.degree().items(), key = lambda x: x[1])[0]
...    return node

rcm = list(cuthill_mckee_ordering(G, heuristic=smallest_degree))

12.5.2 reverse_cuthill_mckee_ordering

reverse_cuthill_mckee_ordering (G, heuristic=None)
Generate an ordering (permutation) of the graph nodes to make a sparse matrix.
Uses the reverse Cuthill-McKee heuristic (based on breadth-first search) [R334].

Parameters
G : graph
A NetworkX graph
heuristic : function, optional
Function to choose starting node for RCM algorithm. If None a node from a psuedo-
peripheral pair is used. A user-defined function can be supplied that takes a graph object
and returns a single node.

Returns
nodes : generator
Generator of nodes in reverse Cuthill-McKee ordering.

See Also:
cuthill_mckee_ordering

Notes
The optimal solution the the bandwidth reduction is NP-complete [R335].

References
[R334], [R335]

Examples

>>> from networkx.utils import reverse_cuthill_mckee_ordering
>>> G = nx.path_graph(4)
>>> rcm = list(reverse_cuthill_mckee_ordering(G))
>>> A = nx.adjacency_matrix(G, nodelist=rcm)

Smallest degree node as heuristic function:

>>> def smallest_degree(G):
...    node, deg = sorted(G.degree().items(), key = lambda x: x[1])[0]
...    return node

rcm = list(reverse_cuthill_mckee_ordering(G, heuristic=smallest_degree))
12.6 Context Managers
reversed(*args, **kwds)  A context manager for temporarily reversing a directed graph in place.

12.6.1 reversed

reversed(*args, **kwds)
A context manager for temporarily reversing a directed graph in place.

This is a no-op for undirected graphs.

Parameters  G : graph
A NetworkX graph.
NetworkX is distributed with the BSD license.

Copyright (C) 2004-2012, NetworkX Developers
Aric Hagberg <hagberg@lanl.gov>
Dan Schult <dschult@colgate.edu>
Pieter Swart <swart@lanl.gov>
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:

* Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above
copyright notice, this list of conditions and the following
 DISCLAIMER in the documentation and/or other materials provided
 with the distribution.

* Neither the name of the NetworkX Developers nor the names of its
 contributors may be used to endorse or promote products derived
 from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
To cite NetworkX please use the following publication:

NetworkX was originally written by Aric Hagberg, Dan Schult, and Pieter Swart, and has been developed 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](http://www.python.org/doc/essays/graphs.html)

Thanks to David Eppstein for the idea of representing a graph $G$ so that “for $n$ in $G$” loops over the nodes in $G$ and $G[n]$ are node $n$’s neighbors.

Thanks to everyone who has improved NetworkX by contributing code, bug reports (and fixes), documentation, and input on design, features, and the future of NetworkX.

Thanks especially to the following contributors:

- Katy Bold contributed the Karate Club graph.
- Hernan Rozenfeld added dorogovtsev_goltsev_mendes_graph and did stress testing.
- Brendt Wohlberg added examples from the Stanford GraphBase.
- Jim Bagrow reported bugs in the search methods.
- Holly Johnsen helped fix the path based centrality measures.
- Arnar Flåtberg fixed the graph laplacian routines.
- Chris Myers suggested using None as a default datatype, suggested improvements for the IO routines, added grid generator index tuple labeling and associated routines, and reported bugs.
- Joel Miller tested and improved the connected components methods fixed bugs and typos in the graph generators, and contributed the random clustered graph generator.
- Keith Briggs sorted out naming issues for random graphs and wrote dense_gnm_random_graph.
- Ignacio Rozada provided the Krapivsky-Redner graph generator.
- Phillipp Pagel helped fix eccentricity etc. for disconnected graphs.
- Sverre Sundsdal contributed bidirectional shortest path and Dijkstra routines, s-metric computation and graph generation
- Ross M. Richardson contributed the expected degree graph generator and helped test the pygraphviz interface.
- Christopher Ellison implemented the VF2 isomorphism algorithm and is a core developer.
- Eben Kenah contributed the strongly connected components and DFS functions.
- Sasha Gutfriend contributed edge betweenness algorithms.
- Udi Weinsberg helped develop intersection and difference operators.
• Matteo Dell’Amico wrote the random regular graph generator.
• Andrew Conway contributed ego_graph, eigenvector centrality, line graph and much more.
• Raf Guns wrote the GraphML writer.
• Salim Fadhley and Matteo Dell’Amico contributed the A* algorithm.
• Fabrice Desclaux contributed the Matplotlib edge labeling code.
• Arpad Horvath fixed the barabasi_albert_graph() generator.
• Minh Van Nguyen contributed the connected_watts_strogatz_graph() and documentation for the Graph and MultiGraph classes.
• Willem Ligtenberg contributed the directed scale free graph generator.
• Loïc Séguin-C. contributed the Ford-Fulkerson max flow and min cut algorithms, and ported all of NetworkX to Python3. He is a NetworkX core developer.
• Paul McGuire improved the performance of the GML data parser.
• Jesus Cerquides contributed the chordal graph algorithms.
• Ben Edwards contributed tree generating functions, the rich club coefficient algorithm, the graph product functions, and a whole lot of other useful nuts and bolts.
• Jon Olav Vik contributed cycle finding algorithms.
• Hugh Brown improved the words.py example from the n^2 algorithm.
• Ben Reilly contributed the shapefile reader and writer.
• Leo Lopes contributed the maximal independent set algorithm.
• Jordi Torrents contributed the bipartite clustering, bipartite node redundancy, square clustering, bipartite projection articulation point, and flow-based connectivity algorithms.
• Dheeraj M R contributed the distance-regular testing algorithm
• Franck Kalala contributed the subgraph_centrality and communicability algorithms
• Simon Knight improved the GraphML functions to handle yEd/yfiles data, and to handle types correctly.
• Conrad Lee contributed the k-clique community finding algorithm.
• Sérgio Nery Simões wrote the function for finding all simple paths, and all shortest paths.
• Robert King contributed union, disjoint union, compose, and intersection operators that work on lists of graphs.
• Nick Mancuso wrote the approximation algorithms for dominating set, edge dominating set, independent set, max clique, and min-weighted vertex cover.
• Brian Cloteaux contributed the linear-time graphicality tests and Havel-Hakimi graph generators
• Alejandro Weinstein contributed the directed Laplacian code
• Dustin Smith wrote the dictionary to numpy array function
• Mathieu Larose sped up the topological sort code
• Vincent Gauthier contributed the Katz centrality algorithm
• Sérgio Nery Simões developed the code for finding all simple paths
**dictionary**  A Python dictionary maps keys to values. Also known as “hashes”, or “associative arrays”. See http://docs.python.org/tutorial/datastructures.html#dictionaries

**ebunch**  An iterable container of edge tuples like a list, iterator, or file.

**edge**  Edges are either two-tuples of nodes (u,v) or three tuples of nodes with an edge attribute dictionary (u,v,dict).

**edge attribute**  Edges can have arbitrary Python objects assigned as attributes by using keyword/value pairs when adding an edge assigning to the G.edge[u][v] attribute dictionary for the specified edge u-v.

**hashable**  An object is hashable if it has a hash value which never changes during its lifetime (it needs a __hash__() method), and can be compared to other objects (it needs an __eq__() or __cmp__() method). Hashable objects which compare equal must have the same hash value.

Hashability makes an object usable as a dictionary key and a set member, because these data structures use the hash value internally.

All of Python’s immutable built-in objects are hashable, while no mutable containers (such as lists or dictionaries) are. Objects which are instances of user-defined classes are hashable by default; they all compare unequal, and their hash value is their id().

Definition from http://docs.python.org/glossary.html

**nbunch**  An nbunch is any iterable container of nodes that is not itself a node in the graph. It can be an iterable or an iterator, e.g. a list, set, graph, file, etc..

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

**node attribute**  Nodes can have arbitrary Python objects assigned as attributes by using keyword/value pairs when adding a node or assigning to the G.node[n] attribute dictionary for the specified node n.
BIBLIOGRAPHY


http://www.inf.uni-konstanz.de/algo/publications/bf-cmbcf-05.pdf


http://www.inf.uni-konstanz.de/algo/publications/bf-cmbcf-05.pdf


http://www.inf.uni-konstanz.de/algo/publications/bf-cmbcf-05.pdf


http://www.inf.uni-konstanz.de/algo/publications/bf-cmbcf-05.pdf


http://en.wikipedia.org/wiki/Tree_decomposition#Treewidth


http://www.cse.msu.edu/~cse835/Papers/Graph_connectivity_revised.pdf

http://www.informatik.uni-augsburg.de/thi/personen/kammer/Graph_Connectivity.pdf


[R228] A model of Internet topology using k-shell decomposition Shai Carmi, Shlomo Havlin, Scott Kirkpatrick, Yuval Shavitt, and Eran Shir, PNAS July 3, 2007 vol. 104 no. 27 11150-11154 http://www.pnas.org/content/104/27/11150.full


http://www.amazon.com/exec/obidos/ASIN/0387948600/ref=ase_thealgorithmrepo/


http://mathworld.wolfram.com/IntersectionArray.html

http://mathworld.wolfram.com/GlobalParameters.html


[R284] Moshen Bayati, Jeong Han Kim, and Amin Saberi, A sequential algorithm for generating random graphs. Algorithmica, Volume 58, Number 4, 860-910, DOI: 10.1007/s00453-009-9340-1


[R322] GEXF graph format, http://gexf.net/format/

[R323] GEXF graph format, http://gexf.net/format/


[R331] http://www.yaml.org


networkx.algorithms.approximation, 127
networkx.algorithms.approximation.clique, 127
networkx.algorithms.approximation.clustering_coefficient, 128
networkx.algorithms.approximation.dominating_set, 128
networkx.algorithms.approximation.independent_set, 130
networkx.algorithms.approximation.matching, 130
networkx.algorithms.approximation.ramsey, 131
networkx.algorithms.approximation.vertex_cover, 131
networkx.algorithms.assortativity, 132
networkx.algorithms.bipartite, 141
networkx.algorithms.bipartite.basic, 143
networkx.algorithms.bipartite.centrality, 158
networkx.algorithms.bipartite.cluster, 153
networkx.algorithms.bipartite.projection, 147
networkx.algorithms.bipartite.redundancy, 157
networkx.algorithms.bipartite.spectral, 152
networkx.algorithms.block, 161
networkx.algorithms.boundary, 162
networkx.algorithms.centrality, 163
networkx.algorithms.chordal.chordal_alg, 184
networkx.algorithms.clique, 187
networkx.algorithms.cluster, 190
networkx.algorithms.community, 193
networkx.algorithms.community.kclique, 193
networkx.algorithms.components, 194
networkx.algorithms.components.attracting, 201
networkx.algorithms.components.biconnected, 203
networkx.algorithms.components.connected, 203
networkx.algorithms.components.coefficient, 194
networkx.algorithms.components.semi-connected, 198
networkx.algorithms.components.strongly_connected, 197
networkx.algorithms.components.weakly_connected, 200
networkx.algorithms.connectivity, 208
networkx.algorithms.connectivity.connectivity, 209
networkx.algorithms.connectivity.cuts, 217
networkx.algorithms.connectivity.stoerwagner, 223
networkx.algorithms.connectivity.util, 225
networkx.algorithms.core, 225
networkx.algorithms.cycles, 229
networkx.algorithms.dag, 231
networkx.algorithms.distance_measures, 234
networkx.algorithms.distance_regular, 236
networkx.algorithms.euler, 237
networkx.algorithms.flow, 239
networkx.algorithms.graphical, 263
networkx.algorithms.hierarchy, 267
networkx.algorithms.isolate, 268
networkx.algorithms.isomorphism, 269
networkx.algorithms.isomorphism.isomorphvf2, 271
networkx.algorithms.link_analysis.hits_alg, 287
networkx.algorithms.link_analysis.pagerank_alg, 282
networkx.algorithms.link_prediction, 289
networkx.algorithms.matching, 296
networkx.algorithms.mis, 297
networkx.algorithms.mst, 298
networkx.algorithms.operators.all, 303
networkx.algorithms.operators.binary, 300
networkx.algorithms.operators.product, 305
networkx.algorithms.operators.unary, 299
networkx.algorithms.richclub, 307
networkx.algorithms.shortest_paths.astar, 324
networkx.algorithms.shortest_paths.dense, 323
networkx.algorithms.shortest_paths.generic, 308
networkx.algorithms.shortest_paths.unweighted, 312
networkx.algorithms.shortest_paths.weighted, 315
networkx.algorithms.simple_paths, 326
networkx.algorithms.swap, 327
networkx.algorithms.traversal.breadth_first_search, 332
networkx.algorithms.traversal.depth_first_search, 328
networkx.algorithms.tree.recognition, 335
networkx.algorithms.vitality, 337

C
networkx.classes.function, 339
networkx.convert, 403
networkx.convert_matrix, 406

d
networkx.drawing.layout, 469
networkx.drawing.nx_agraph, 464
networkx.drawing.nx_pydot, 467
networkx.drawing.nx_pydot, 455

e
networkx.exception, 475

g
networkx.generators.atlas, 347
networkx.generators.bipartite, 379
networkx.generators.classic, 347
networkx.generators.degree_seq, 365
networkx.generators.directed, 372
networkx.generators.ego, 385
networkx.generators.geometric, 375
networkx.generators.hybrid, 379
networkx.generators.intersection, 386
networkx.generators.line, 383

networkx.generators.random_clustered, 371
networkx.generators.random_graphs, 356
networkx.generators.small, 352
networkx.generators.social, 388
networkx.generators.stochastic, 386

networkx.linalg.algebraicconnectivity, 394
networkx.linalg.attrmatrix, 397
networkx.linalg.graphmatrix, 389
networkx.linalg.laplacianmatrix, 391
networkx.linalg.spectrum, 393

networkx.readwrite.adjlist, 415
networkx.readwrite.edgelist, 422
networkx.readwrite.gexf, 429
networkx.readwrite.gml, 431
networkx.readwrite.gpickle, 434
networkx.readwrite.graph6, 444
networkx.readwrite.graphml, 435

networkx.readwrite.json_graph, 437
networkx.readwrite.leda, 442
networkx.readwrite multiline_adjlist, 419
networkx.readwrite.nx_shp, 452
networkx.readwrite.nx_yaml, 443
networkx.readwrite.pajek, 450
networkx.readwrite.sparse6, 447

U
networkx.utils, 477
networkx.utils.contextmanagers, 484
networkx.utils.decorators, 481
networkx.utils.misc, 477
networkx.utils.random_sequence, 478
networkx.utils.rcm, 482
networkx.utils.union_find, 478
Symbols

__contains__() (DiGraph method), 56
__contains__() (Graph method), 27
__contains__() (MultiDiGraph method), 116
__contains__() (MultiGraph method), 86
getitem() (DiGraph method), 52
getitem() (Graph method), 25
getitem() (MultiDiGraph method), 113
ggetitem() (MultiGraph method), 84
init() (DiGraph method), 38
init() (DiGraphMatcher method), 275
init() (Graph method), 12
init() (GraphMatcher method), 273
init() (MultiDiGraph method), 97
init() (MultiGraph method), 69
__iter__() (DiGraph method), 48
__iter__() (Graph method), 21
__iter__() (MultiDiGraph method), 108
__iter__() (MultiGraph method), 80
__len__() (DiGraph method), 57
__len__() (Graph method), 29
__len__() (MultiDiGraph method), 118
__len__() (MultiGraph method), 88

A

adamic_adar_index() (in module networkx.algorithms.link_prediction), 291
add_cycle() (DiGraph method), 45
add_cycle() (Graph method), 19
add_cycle() (MultiDiGraph method), 105
add_cycle() (MultiGraph method), 77
add_edge() (DiGraph method), 41
add_edge() (Graph method), 15
add_edge() (MultiDiGraph method), 101
add_edge() (MultiGraph method), 72
add_edges_from() (DiGraph method), 42
add_edges_from() (Graph method), 16
add_edges_from() (MultiDiGraph method), 102
add_edges_from() (MultiGraph method), 73
add_node() (DiGraph method), 39
add_node() (Graph method), 12
add_nodes_from() (MultiDiGraph method), 98
add_nodes_from() (MultiGraph method), 69
add_nodes_from() (MultiDiGraph method), 39
add_nodes_from() (MultiGraph method), 13
add_nodes_from() (MultiDiGraph method), 99
add_nodes_from() (MultiGraph method), 70
add_path() (DiGraph method), 45
add_path() (Graph method), 19
add_path() (MultiDiGraph method), 105
add_path() (MultiGraph method), 76
add_star() (DiGraph method), 44
add_star() (Graph method), 18
add_star() (MultiDiGraph method), 105
add_star() (MultiGraph method), 76
add_weighted_edges_from() (DiGraph method), 43
add_weighted_edges_from() (Graph method), 17
add_weighted_edges_from() (MultiDiGraph method), 102
add_weighted_edges_from() (MultiGraph method), 74
adjacency_data() (in module networkx.readwrite.json_graph), 439
adjacency_graph() (in module networkx.readwrite.json_graph), 440
adjacency_iter() (DiGraph method), 54
adjacency_iter() (Graph method), 26
adjacency_iter() (MultiDiGraph method), 114
adjacency_iter() (MultiGraph method), 85
adjacency_list() (DiGraph method), 53
adjacency_list() (Graph method), 25
adjacency_list() (MultiDiGraph method), 114
adjacency_list() (MultiGraph method), 84
adjacency_matrix() (in module networkx.linalg.graphmatrix), 389
adjacency_spectrum() (in module networkx.linalg.spectrum), 394
algebraic_connectivity() (in module networkx.linalg.algebraicconnectivity), 394
all_neighbors() (in module networkx.classes.function), 341
all_pairs_dijkstra_path() (in module networkx.algorithms.shortest_paths.weighted), 318
all_pairs_dijkstra_path_length() (in module networkx.algorithms.shortest_paths.weighted), 318
all_pairs_node_connectivity() (in module networkx.algorithms.connectivity.connectivity), 209
all_pairs_shortest_path() (in module networkx.algorithms.shortest_paths.unweighted), 313
all_pairs_shortest_path_length() (in module networkx.algorithms.shortest_paths.unweighted), 314
all_shortest_paths() (in module networkx.algorithms.shortest_paths.generic), 309
all_simple_paths() (in module networkx.algorithms.simple_paths), 326
ancestors() (in module networkx.algorithms.dag), 231
approximate_current_flow_betweenness_centrality() (in module networkx.algorithms.centrality), 171
articulation_points() (in module networkx.algorithms.components.biconnected), 207
astar_path() (in module networkx.algorithms.shortest_paths.astar), 324
astar_path_length() (in module networkx.algorithms.shortest_paths.astar), 325
attr_matrix() (in module networkx.linalg.attrmatrix), 397
attr_sparse_matrix() (in module networkx.linalg.attrmatrix), 399
attracting_component_subgraphs() (in module networkx.algorithms.components.attracting), 202
attracting_components() (in module networkx.algorithms.components.attracting), 202
attribute_assortativity_coefficient() (in module networkx.algorithms.assortativity), 133
attribute_mixing_dict() (in module networkx.algorithms.assortativity), 141
attribute_mixing_matrix() (in module networkx.algorithms.assortativity), 139
authority_matrix() (in module networkx.algorithms.link_analysis.hits_alg), 289
average_clustering() (in module networkx.algorithms.approximation.clustering_coefficient), 128
average_clustering() (in module networkx.algorithms.bipartite.cluster), 154
average_clustering() (in module networkx.algorithms.cluster), 192
average_degree_connectivity() (in module networkx.algorithms.assortativity), 137
average_neighbor_degree() (in module networkx.algorithms.assortativity), 136
average_node_connectivity() (in module networkx.algorithms.connectivity.connectivity), 209
average_shortest_path_length() (in module networkx.algorithms.shortest_paths.generic), 311

B
balanced_tree() (in module networkx.generators.classic), 348
barabasi_albert_graph() (in module networkx.generators.random_graphs), 362
barbell_graph() (in module networkx.generators.classic), 348
bellman_ford() (in module networkx.algorithms.shortest_paths.weighted), 321
betweenness_centrality() (in module networkx.algorithms.bipartite.centrality), 160
betweenness_centrality() (in module networkx.algorithms.bipartite.centrality), 166
bfs_edges() (in module networkx.algorithms.traversal.breadth_first_search), 332
bfs_predecessors() (in module networkx.algorithms.traversal.breadth_first_search), 334
bfs_successors() (in module networkx.algorithms.traversal.breadth_first_search), 334
bfs_tree() (in module networkx.algorithms.traversal.breadth_first_search), 333
biadjacency_matrix() (in module networkx.algorithms.bipartite.basic), 146
biconnected_component_edges() (in module networkx.algorithms.components.biconnected), 205
biconnected_component_subgraphs() (in module networkx.algorithms.components.biconnected), 206
biconnected_components() (in module networkx.algorithms.components.biconnected), 204
bidirectional_dijkstra() (in module networkx.algorithms.shortest_paths.weighted), 320
binomial_graph() (in module networkx.generators.random_graphs), 359
bipartite_alternating_havel_hakimi_graph() (in module networkx.generators.bipartite), 381
Index

bipartite_configuration_model() (in module networkx.generators.bipartite), 379
bipartite_gnmk_random_graph() (in module networkx.generators.bipartite), 383
bipartite_havel_hakimi_graph() (in module networkx.generators.bipartite), 380
bipartite_preferential_attachment_graph() (in module networkx.generators.bipartite), 382
bipartite_random_graph() (in module networkx.generators.bipartite), 382
bipartite_reverse_havel_hakimi_graph() (in module networkx.generators.bipartite), 381
blockmodel() (in module networkx.algorithms.block), 161
build_auxiliary_edge_connectivity() (in module networkx.algorithms.connectivity.utils), 225
build_auxiliary_node_connectivity() (in module networkx.algorithms.connectivity.utils), 225
build_residual_network() (in module networkx.algorithms.flow), 254
bull_graph() (in module networkx.generators.small), 353

candidate_pairs_iter() (DiGraphMatcher method), 276
candidate_pairs_iter() (GraphMatcher method), 274
capacity_scaling() (in module networkx.algorithms.flow), 261
cartesian_product() (in module networkx.algorithms.operators.product), 305
categorical_edge_match() (in module networkx.algorithms.isomorphism), 277
categorical_multiedge_match() (in module networkx.algorithms.isomorphism), 278
categorical_node_match() (in module networkx.algorithms.isomorphism), 277
center() (in module networkx.algorithms.distance_measures), 234
chordal_graph_cliques() (in module networkx.algorithms.chordal.chordal_alg), 185
chordal_graph_treewidth() (in module networkx.algorithms.chordal.chordal_alg), 186
chvatal_graph() (in module networkx.generators.small), 353
circular_ladder_graph() (in module networkx.generators.classic), 349
circular_layout() (in module networkx.drawing.layout), 470
clear() (DiGraph method), 46
clear() (Graph method), 20
clear() (MultiDiGraph method), 106
clear() (MultiGraph method), 77
clique_removal() (in module networkx.algorithms.approximation.clique), 128
clicks_containing_node() (in module networkx.algorithms.clique), 189
closeness_centrality() (in module networkx.algorithms.centrality), 165
closeness_vitality() (in module networkx.algorithms.vitality), 337
clustering() (in module networkx.algorithms.bipartite.cluster), 153
clustering() (in module networkx.algorithms.cluster), 191
cn_soundarajan_hopcroft() (in module networkx.algorithms.link_prediction), 293
collaboration_weighted_projected_graph() (in module networkx.algorithms.projection), 149
color() (in module networkx.algorithms.bipartite.basic), 144
common_neighbors() (in module networkx.classes.function), 342
communicability() (in module networkx.algorithms.centrality), 177
communicability_betweenness_centrality() (in module networkx.algorithms.centrality), 181
communicability_centrality() (in module networkx.algorithms.centrality), 179
communicability_centrality_exp() (in module networkx.algorithms.centrality), 180
complement() (in module networkx.algorithms.operators.unary), 299
complete_bipartite_graph() (in module networkx.generators.classic), 349
complete_graph() (in module networkx.generators.classic), 349
compose() (in module networkx.algorithms.operators.binary), 300
compose_all() (in module networkx.algorithms.operators.all), 303
condensation() (in module networkx.algorithms.components.strongly_connected), 200
configuration_model() (in module networkx.generators.degree_seq), 365
connected_component_subgraphs() (in module networkx.algorithms.components.connected), 196
connected_components() (in module networkx.algorithms.components.connected), 195
connected_double_edge_swap() (in module networkx.algorithms.swap), 328
connected_watts_strogatz_graph() (in module networkx.generators.random_graphs), 361

[383x748]NetworkX Reference, Release 1.9
[72x710]bipartite_configuration_model() (in module net-
workx.generators.bipartite), 379
bipartite_gnmk_random_graph() (in module net-
workx.generators.bipartite), 383
bipartite_havel_hakimi_graph() (in module net-
workx.generators.bipartite), 380
bipartite_preferential_attachment_graph() (in module
networkx.generators.bipartite), 382
bipartite_random_graph() (in module net-
workx.generators.bipartite), 382
bipartite_reverse_havel_hakimi_graph() (in module net-
workx.generators.bipartite), 381
blockmodel() (in module networkx.algorithms.block), 161
build_auxiliary_edge_connectivity() (in module net-
workx.algorithms.connectivity.utils), 225
build_auxiliary_node_connectivity() (in module net-
workx.algorithms.connectivity.utils), 225
build_residual_network() (in module net-
workx.algorithms.flow), 254
bull_graph() (in module networkx.generators.small), 353

candidate_pairs_iter() (DiGraphMatcher method), 276
candidate_pairs_iter() (GraphMatcher method), 274
capacity_scaling() (in module networkx.algorithms.flow), 261
cartesian_product() (in module networkx.algorithms.operators.product), 305
categorical_edge_match() (in module networkx.algorithms.isomorphism), 277
categorical_multiedge_match() (in module networkx.algorithms.isomorphism), 278
categorical_node_match() (in module networkx.algorithms.isomorphism), 277
center() (in module networkx.algorithms.distance_measures), 234
chordal_graph_cliques() (in module networkx.algorithms.chordal.chordal_alg), 185
chordal_graph_treewidth() (in module networkx.algorithms.chordal.chordal_alg), 186
chvatal_graph() (in module networkx.generators.small), 353
circular_ladder_graph() (in module networkx.generators.classic), 349
circular_layout() (in module networkx.drawing.layout), 470
clear() (DiGraph method), 46
clear() (Graph method), 20
clear() (MultiDiGraph method), 106
clear() (MultiGraph method), 77
clique_removal() (in module networkx.algorithms.approximation.clique), 128
clicks_containing_node() (in module networkx.algorithms.clique), 189
closeness_centrality() (in module networkx.algorithms.centrality), 165
closeness_vitality() (in module networkx.algorithms.vitality), 337
clustering() (in module networkx.algorithms.bipartite.cluster), 153
clustering() (in module networkx.algorithms.cluster), 191
cn_soundarajan_hopcroft() (in module networkx.algorithms.link_prediction), 293
collaboration_weighted_projected_graph() (in module networkx.algorithms.projection), 149
color() (in module networkx.algorithms.bipartite.basic), 144
common_neighbors() (in module networkx.classes.function), 342
communicability() (in module networkx.algorithms.centrality), 177
communicability_betweenness_centrality() (in module networkx.algorithms.centrality), 181
communicability_centrality() (in module networkx.algorithms.centrality), 179
communicability_centrality_exp() (in module networkx.algorithms.centrality), 180
complement() (in module networkx.algorithms.operators.unary), 299
complete_bipartite_graph() (in module networkx.generators.classic), 349
complete_graph() (in module networkx.generators.classic), 349
compose() (in module networkx.algorithms.operators.binary), 300
compose_all() (in module networkx.algorithms.operators.all), 303
condensation() (in module networkx.algorithms.components.strongly_connected), 200
configuration_model() (in module networkx.generators.degree_seq), 365
connected_component_subgraphs() (in module networkx.algorithms.components.connected), 196
connected_components() (in module networkx.algorithms.components.connected), 195
connected_double_edge_swap() (in module networkx.algorithms.swap), 328
connected_watts_strogatz_graph() (in module networkx.generators.random_graphs), 361

Index 509
copy() (DiGraph method), 63
copy() (Graph method), 33
copy() (MultiDiGraph method), 124
copy() (MultiGraph method), 92
core_number() (in module networkx.algorithms.core), 226
cost_of_flow() (in module networkx.algorithms.flow), 259
could_be_isomorphic() (in module networkx.algorithms.isomorphism), 271
create_degree_sequence() (in module networkx.utils.random_sequence), 479
create_empty_copy() (in module networkx.classes.function), 340
cubical_graph() (in module networkx.generators.small), 353
cumulative_distribution() (in module networkx.utils.random_sequence), 479
cumulative_sum() (in module networkx.utils.misc), 478
current_flow_betweenness_centrality() (in module networkx.algorithms.centrality), 169
current_flow_closeness_centrality() (in module networkx.algorithms.centrality), 168
cuthill_mckee_ordering() (in module networkx.utils.rcm), 482
cycle_basis() (in module networkx.algorithms.cycles), 229
cycle_graph() (in module networkx.generators.classic), 349
davis_southern_women_graph() (in module networkx.generators.social), 388
default_opener() (in module networkx.utils.misc), 478
degree() (DiGraph method), 57
degree() (Graph method), 29
degree() (in module networkx.classes.function), 339
degree() (MultiDiGraph method), 118
degree() (MultiGraph method), 88
degree_assortativity_coefficient() (in module networkx.algorithms.assortativity), 132
degree_centrality() (in module networkx.algorithms.bipartite.centrality), 159
degree_centrality() (in module networkx.algorithms.centrality), 163
degree_histogram() (in module networkx.classes.function), 339
degree_iter() (DiGraph method), 58
degree_iter() (Graph method), 30
degree_iter() (MultiDiGraph method), 118
degree_iter() (MultiGraph method), 89
degree_mixing_dict() (in module networkx.algorithms.assortativity), 140
degree_mixing_matrix() (in module networkx.algorithms.assortativity), 140
degree_pearson_correlation_coefficient() (in module networkx.algorithms.assortativity), 135
degree_sequence_tree() (in module networkx.generators.degree_seq), 370
degrees() (in module networkx.algorithms.bipartite.basic), 145
dense_gnm_random_graph() (in module networkx.generators.random_graphs), 358
density() (in module networkx.algorithms.bipartite.basic), 145
density() (in module networkx.classes.function), 340
desargues_graph() (in module networkx.generators.small), 354
desendants() (in module networkx.algorithms.dag), 231
dfs_edges() (in module networkx.algorithms.traversal.depth_first_search), 328
dfs_labeled_edges() (in module networkx.algorithms.traversal.depth_first_search), 332
dfs_postorder_nodes() (in module networkx.algorithms.traversal.depth_first_search), 331
dfs_predecessors() (in module networkx.algorithms.traversal.depth_first_search), 329
dfs_preorder_nodes() (in module networkx.algorithms.traversal.depth_first_search), 331
dfs_successors() (in module networkx.algorithms.traversal.depth_first_search), 330
dfs_tree() (in module networkx.algorithms.traversal.depth_first_search), 329
diameter() (in module networkx.algorithms.distance_measures), 234
diamond_graph() (in module networkx.generators.small), 354
dictionary, 493
difference() (in module networkx.algorithms.operators.binary), 302
DiGraph() (in module networkx), 35
dijkstra_path() (in module networkx.algorithms.shortest_paths.weighted), 315
dijkstra_path_length() (in module networkx.algorithms.shortest_paths.weighted), 316
dijkstra_predecessor_and_distance() (in module networkx.algorithms.shortest_paths.weighted), 321
NetworkX Reference, Release 1.9

directed_configuration_model() (in module networkx.generators.degree_seq), 367
directed_havel_hakimi_graph() (in module networkx.generators.degree_seq), 369
directed_laplacian_matrix() (in module networkx.linalg.laplacianmatrix), 392
discrete_sequence() (in module networkx.utils.random_sequence), 479
disjoint_union() (in module networkx.algorithms.operators.binary), 301
disjoint_union_all() (in module networkx.algorithms.operators.all), 304
dispersion() (in module networkx.algorithms.centrality), 184
dodecahedral_graph() (in module networkx.generators.small), 354
dorogovtsev_goltsev_mendes_graph() (in module networkx.generators.classic), 349
double_edge_swap() (in module networkx.algorithms.swap), 327
draw() (in module networkx.drawing.nx_pylab), 455
draw_circular() (in module networkx.drawing.nx_pylab), 463
draw_graphviz() (in module networkx.drawing.nx_pylab), 464
draw_networkx() (in module networkx.drawing.nx_pylab), 456
draw_networkx_edge_labels() (in module networkx.drawing.nx_pylab), 462
draw_networkx_edges() (in module networkx.drawing.nx_pylab), 460
draw_networkx_labels() (in module networkx.drawing.nx_pylab), 461
draw_networkx_nodes() (in module networkx.drawing.nx_pylab), 458
draw_random() (in module networkx.drawing.nx_pylab), 463
draw_shell() (in module networkx.drawing.nx_pylab), 464
draw_spectral() (in module networkx.drawing.nx_pylab), 463
draw_spring() (in module networkx.drawing.nx_pylab), 464
deedge_connectivity() (in module networkx.algorithms.connectivity.connectivity), 210
deedge_current_flow_betweenness_centrality() (in module networkx.algorithms.centrality), 170
deedge_load() (in module networkx.algorithms.centrality), 183
dedges() (DiGraph method), 48
dedges() (Graph method), 22
dedges() (in module networkx.classes.function), 342
dedges() (MultiDiGraph method), 108
dedges() (MultiGraph method), 80
dedges_iter() (DiGraph method), 49
dedges_iter() (Graph method), 22
dedges_iter() (in module networkx.classes.function), 343
dedges_iter() (MultiDiGraph method), 109
dedges_iter() (MultiGraph method), 81
edmonds_karp() (in module networkx.algorithms.flow), 247
ego_graph() (in module networkx.generators.ego), 385
eigenvector_centrality() (in module networkx.algorithms.centrality), 173
eigenvector_centrality_numpy() (in module networkx.algorithms.centrality), 174
eempty_graph() (in module networkx.generators.classic), 350
erdos_renyi_graph() (in module networkx.generators.random_graphs), 359
estrada_index() (in module networkx.algorithms.centrality), 182
eulerian_circuit() (in module networkx.algorithms.euler), 238
expected_degree_graph() (in module networkx.generators.degree_seq), 368

F
fast_could_be_isomorphic() (in module networkx.algorithms.isomorphism), 271
fast_gnp_random_graph() (in module networkx.generators.random_graphs), 356
faster_could_be_isomorphic() (in module networkx.algorithms.isomorphism), 271
fiedler_vector() (in module networkx.linalg.algebraicconnectivity), 395
find_cliques() (in module networkx.algorithms.clique), 188
find_induced_nodes() (in module networkx.algorithms.clique), 189
flatten() (in module networkx.utils.misc), 477
florentine_families_graph() (in module networkx.generators.social), 388
flow_hierarchy() (in module networkx.algorithms.hierarchy), 267

E
ebunch, 493
eccentricity() (in module networkx.algorithms.distance_measures), 235
edge, 493
edge attribute, 493
edge_betweenness_centrality() (in module networkx.algorithms.centrality), 167
edge_boundary() (in module networkx.algorithms.centrality), 162

Index 511
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Module/Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>floyd_warshall()</td>
<td>networkx.algorithms.shortest_paths.dense</td>
<td>323</td>
</tr>
<tr>
<td>floyd_warshall_numpy()</td>
<td>networkx.algorithms.shortest_paths.dense</td>
<td>324</td>
</tr>
<tr>
<td>floyd_warshall_predecessor_and_distance()</td>
<td>networkx.algorithms.shortest_paths.dense</td>
<td>323</td>
</tr>
<tr>
<td>ford_fulkerson()</td>
<td>networkx.algorithms.flow</td>
<td>248</td>
</tr>
<tr>
<td>freeze()</td>
<td>networkx.classes.function</td>
<td>345</td>
</tr>
<tr>
<td>from_agraph()</td>
<td>networkx.drawing.nx_agraph</td>
<td>465</td>
</tr>
<tr>
<td>from_dict_of_dicts()</td>
<td>networkx.convert</td>
<td>404</td>
</tr>
<tr>
<td>from_dict_of_lists()</td>
<td>networkx.convert</td>
<td>405</td>
</tr>
<tr>
<td>from_edgelist()</td>
<td>networkx.convert</td>
<td>406</td>
</tr>
<tr>
<td>from_numpy_matrix()</td>
<td>networkx.convert_matrix</td>
<td>410</td>
</tr>
<tr>
<td>from_pydot()</td>
<td>networkx.drawing.nx_pydot</td>
<td>468</td>
</tr>
<tr>
<td>from_scipy_sparse_matrix()</td>
<td>networkx.convert_matrix</td>
<td>412</td>
</tr>
<tr>
<td>frucht_graph()</td>
<td>networkx.generators.small</td>
<td>354</td>
</tr>
<tr>
<td>get_edge_attributes()</td>
<td>networkx.classes.function</td>
<td>345</td>
</tr>
<tr>
<td>get_edge_data() (DiGraph method)</td>
<td>networkx.algorithms.shortest_paths.generic</td>
<td>512</td>
</tr>
<tr>
<td>get_edge_data() (Graph method)</td>
<td>networkx.algorithms.shortest_paths.generic</td>
<td>23</td>
</tr>
<tr>
<td>get_edge_data() (MultiDiGraph method)</td>
<td>networkx.algorithms.shortest_paths.generic</td>
<td>112</td>
</tr>
<tr>
<td>get_edge_data() (MultiGraph method)</td>
<td>networkx.algorithms.shortest_paths.generic</td>
<td>82</td>
</tr>
<tr>
<td>get_node_attributes() (DiGraph method)</td>
<td>networkx.classes.function</td>
<td>344</td>
</tr>
<tr>
<td>global_parameters()</td>
<td>networkx.algorithms.distance_regular</td>
<td>237</td>
</tr>
<tr>
<td>kn_graph()</td>
<td>networkx.generators.directed</td>
<td>373</td>
</tr>
<tr>
<td>gnc_graph()</td>
<td>networkx.generators.directed</td>
<td>374</td>
</tr>
<tr>
<td>gnm_random_graph()</td>
<td>networkx.generators.random_graphs</td>
<td>358</td>
</tr>
<tr>
<td>gnp_random_graph()</td>
<td>networkx.generators.random_graphs</td>
<td>357</td>
</tr>
<tr>
<td>gnr_graph()</td>
<td>networkx.generators.directed</td>
<td>373</td>
</tr>
<tr>
<td>google_matrix()</td>
<td>networkx.algorithms.link_analysis.pagerank_alg</td>
<td>286</td>
</tr>
<tr>
<td>Graph()</td>
<td>networkx</td>
<td>9</td>
</tr>
<tr>
<td>graph_atlas_g()</td>
<td>networkx.generators.atlas</td>
<td>347</td>
</tr>
<tr>
<td>graph_clique_number()</td>
<td>networkx.algorithms.clique</td>
<td>189</td>
</tr>
<tr>
<td>graph_number_of_cliques()</td>
<td>networkx.algorithms.clique</td>
<td>189</td>
</tr>
<tr>
<td>graphviz_layout()</td>
<td>networkx.drawing.nx_agraph</td>
<td>466</td>
</tr>
<tr>
<td>graphviz_layout()</td>
<td>networkx.drawing.nx_pydot</td>
<td>469</td>
</tr>
<tr>
<td>grid_2d_graph()</td>
<td>networkx.generators.classic</td>
<td>350</td>
</tr>
<tr>
<td>grid_graph()</td>
<td>networkx.generators.classic</td>
<td>350</td>
</tr>
<tr>
<td>has_edge() (DiGraph method)</td>
<td>networkx.algorithms.isomorphism</td>
<td>281</td>
</tr>
<tr>
<td>has_edge() (Graph method)</td>
<td>networkx.algorithms.isomorphism</td>
<td>281</td>
</tr>
<tr>
<td>has_edge() (MultiDiGraph method)</td>
<td>networkx.algorithms.isomorphism</td>
<td>281</td>
</tr>
<tr>
<td>has_edge() (MultiGraph method)</td>
<td>networkx.algorithms.isomorphism</td>
<td>281</td>
</tr>
<tr>
<td>has_node() (DiGraph method)</td>
<td>networkx.utils.misc</td>
<td>478</td>
</tr>
<tr>
<td>has_node() (Graph method)</td>
<td>networkx.utils.misc</td>
<td>478</td>
</tr>
<tr>
<td>has_node() (MultiDiGraph method)</td>
<td>networkx.utils.misc</td>
<td>478</td>
</tr>
<tr>
<td>has_node() (MultiGraph method)</td>
<td>networkx.utils.misc</td>
<td>478</td>
</tr>
<tr>
<td>has_path() (DiGraph method)</td>
<td>networkx.algorithms.shortest_paths.generic</td>
<td>312</td>
</tr>
<tr>
<td>has_path() (Graph method)</td>
<td>networkx.algorithms.shortest_paths.generic</td>
<td>312</td>
</tr>
<tr>
<td>has_path() (MultiDiGraph method)</td>
<td>networkx.algorithms.shortest_paths.generic</td>
<td>312</td>
</tr>
<tr>
<td>has_path() (MultiGraph method)</td>
<td>networkx.algorithms.shortest_paths.generic</td>
<td>312</td>
</tr>
<tr>
<td>hashable</td>
<td>networkx</td>
<td>493</td>
</tr>
<tr>
<td>havel_hakimi_graph()</td>
<td>networkx.generators.degree_seq</td>
<td>369</td>
</tr>
<tr>
<td>heawood_graph()</td>
<td>networkx.generators.small</td>
<td>354</td>
</tr>
</tbody>
</table>
Index
K
k_clique_communities() (in module networkx.algorithms.community.kclique), 194
k_core() (in module networkx.algorithms.core), 226
k_corona() (in module networkx.algorithms.core), 229
k_crust() (in module networkx.algorithms.core), 228
k_nearest_neighbors() (in module networkx.algorithms.assortativity), 138
k_random_intersection_graph() (in module networkx.generators.intersection), 387
k_shell() (in module networkx.algorithms.core), 227
karate_club_graph() (in module networkx.generators.social), 388
katz_centrality() (in module networkx.algorithms.centrality), 174
katz_centrality_numpy() (in module networkx.algorithms.centrality), 176
kl_connected_subgraph() (in module networkx.generators.hybrid), 379
kosaraju_strongly_connected_components() (in module networkx.algorithms.components.strongly_connected), 199
krackhardt_kite_graph() (in module networkx.generators.small), 355

L
ladder_graph() (in module networkx.generators.classic), 351
laplacian_matrix() (in module networkx.linalg.laplacianmatrix), 391
laplacian_spectrum() (in module networkx.linalg.spectrum), 393
latapy_clustering() (in module networkx.algorithms.bipartite.cluster), 155
LCF_graph() (in module networkx.generators.small), 353
lexicographic_product() (in module networkx.algorithms.operators.product), 305
line_graph() (in module networkx.generators.line), 384
load_centrality() (in module networkx.algorithms.core.labelling.centrality), 183
local_edge_connectivity() (in module networkx.algorithms.connectivity.connectivity), 211
local_node_connectivity() (in module networkx.algorithms.connectivity.connectivity), 213
lollipop_graph() (in module networkx.generators.classic), 351

M
make_clique_bipartite() (in module networkx.algorithms.clique), 189
make_max_clique_graph() (in module networkx.algorithms.clique), 188
make_small_graph() (in module networkx.generators.core), 352
make_str() (in module networkx.utils.misc), 478
match() (DiGraphMatcher method), 276
match() (GraphMatcher method), 275
max_clique() (in module networkx.algorithms.approximation.clique), 127
max_flow_min_cost() (in module networkx.algorithms.flow), 260
max_weight_matching() (in module networkx.algorithms.matching), 296
maximal_independent_set() (in module networkx.algorithms.mis), 297
maximal_matching() (in module networkx.algorithms.matching), 296
maximum_flow() (in module networkx.algorithms.flow), 239
maximum_flow_value() (in module networkx.algorithms.flow), 241
maximum_independent_set() (in module networkx.algorithms.approximation.independent_set), 130
min_cost_flow() (in module networkx.algorithms.flow), 258
min_cost_flow_cost() (in module networkx.algorithms.flow), 257
min_edge_dominating_set() (in module networkx.algorithms.approximation.dominating_set), 129
min_maximal_matching() (in module networkx.algorithms.approximation.matching), 131
min_weighted_dominating_set() (in module networkx.algorithms.approximation.dominating_set), 129
min_weighted_vertex_cover() (in module networkx.algorithms.approximation.vertex_cover), 132
minimum_cut() (in module networkx.algorithms.flow), 243
minimum_cut_value() (in module networkx.algorithms.flow), 245
minimum_edge_cut() (in module networkx.algorithms.connectivity.cuts), 217
minimum_node_cut() (in module networkx.algorithms.connectivity.cuts), 218
minimum_spanning_edges() (in module networkx.algorithms.mst), 299
minimum_spanning_tree() (in module networkx.algorithms.mst), 298
minimum_st_edge_cut() (in module networkx.algorithms.mst), 298
workx.algorithms.connectivity.cuts), 220
minimum_st_node_cut() (in module networkx.algorithms.connectivity.cuts), 222
moebius_kantor_graph() (in module networkx.generators.small), 355
MultiDiGraph() (in module networkx), 95
MultiGraph() (in module networkx), 66

N

 navigable_small_world_graph() (in module networkx.generators.geometric), 378
nbunch, 493
nbunch_iter() (DiGraph method), 54
nbunch_iter() (Graph method), 26
nbunch_iter() (MultiDiGraph method), 115
nbunch_iter() (MultiGraph method), 85
negative_edge_cycle() (in module networkx.algorithms.shortest_paths.weighted), 322
neighbors() (DiGraph method), 52
neighbors() (Graph method), 24
neighbors() (MultiDiGraph method), 113
neighbors() (MultiGraph method), 83
neighbors_iter() (DiGraph method), 52
neighbors_iter() (Graph method), 24
neighbors_iter() (MultiDiGraph method), 113
neighbors_iter() (MultiGraph method), 83
network_simplex() (in module networkx.algorithms.shortest_paths.weighted), 255
networkx.algorithms.approximation (module), 127
networkx.algorithms.approximation.clique (module), 127
networkx.algorithms.approximation.clustering_coefficient (module), 128
networkx.algorithms.approximation.dominating_set (module), 128
networkx.algorithms.approximation.independent_set (module), 130
networkx.algorithms.approximation.matching (module), 130
networkx.algorithms.approximation.ramsey (module), 131
networkx.algorithms.approximation.vertex_cover (module), 131
networkx.algorithms.assortativity (module), 132
networkx.algorithms.bipartite (module), 141
networkx.algorithms.bipartite.basic (module), 143
networkx.algorithms.bipartite.centrality (module), 158
networkx.algorithms.bipartite.cluster (module), 153
networkx.algorithms.bipartite.projection (module), 147
networkx.algorithms.bipartite.redundancy (module), 157
networkx.algorithms.bipartite.spectral (module), 152
networkx.algorithms.block (module), 161
networkx.algorithms.boundary (module), 162
networkx.algorithms.centrality (module), 163
networkx.algorithms.chordal.chordal_alg (module), 184
networkx.algorithms.cluster (module), 187
networkx.algorithms.community (module), 190
networkx.algorithms.community.kclique (module), 193
networkx.algorithms.components (module), 194
networkx.algorithms.components.attracting (module), 201
networkx.algorithms.components.biconnected (module), 203
networkx.algorithms.components.connected (module), 194
networkx.algorithms.components.semicomponents (module), 208
networkx.algorithms.components.strongly_connected (module), 197
networkx.algorithms.components.weakly_connected (module), 200
networkx.algorithms.connectivity (module), 208
networkx.algorithms.connectivity.cuts (module), 217
networkx.algorithms.connectivity.connectivity (module), 209
networkx.algorithms.connectivity.cuts (module), 217
networkx.algorithms.connectivity.stoerwagner (module), 223
networkx.algorithms.connectivity.utils (module), 225
networkx.algorithms.core (module), 225
networkx.algorithms.cycles (module), 229
networkx.algorithms.dag (module), 231
networkx.algorithms.distance_measures (module), 234
networkx.algorithms.distance_regular (module), 236
networkx.algorithms.euler (module), 237
networkx.algorithms.flow (module), 239
networkx.algorithms.graphical (module), 263
networkx.algorithms.hierarchy (module), 267
networkx.algorithms.isolate (module), 268
networkx.algorithms.isomorphism (module), 269
networkx.algorithms.isomorphism.isomorphvf2 (module), 271
networkx.algorithms.link_analysis.hits_alg (module), 287
networkx.algorithms.link_analysis.pagerank_alg (module), 282
networkx.algorithms.link_prediction (module), 289
networkx.algorithms.matching (module), 296
networkx.algorithms.mis (module), 297
networkx.algorithms.mst (module), 298
networkx.algorithms.operators.all (module), 303
networkx.algorithms.operators.binary (module), 300
networkx.algorithms.operators.product (module), 305
networkx.algorithms.operators.unary (module), 307
networkx.algorithms.shortest_paths.astar (module), 324
networkx.algorithms.shortest_paths.dense (module), 323
<table>
<thead>
<tr>
<th>NetworkX Reference, Release 1.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>networkx.algorithms.shortest_paths.generic (module), 308</td>
</tr>
<tr>
<td>networkx.algorithms.shortest_paths.unweighted (module), 312</td>
</tr>
<tr>
<td>networkx.algorithms.shortest_paths.weighted (module), 315</td>
</tr>
<tr>
<td>networkx.algorithms.simple_paths (module), 326</td>
</tr>
<tr>
<td>networkx.algorithms.swap (module), 327</td>
</tr>
<tr>
<td>networkx.algorithms.traversal.breadth_first_search (module), 332</td>
</tr>
<tr>
<td>networkx.algorithms.traversal.depth_first_search (module), 328</td>
</tr>
<tr>
<td>networkx.algorithms.tree.recognition (module), 335</td>
</tr>
<tr>
<td>networkx.algorithms.vitality (module), 337</td>
</tr>
<tr>
<td>networkx.classes.function (module), 339</td>
</tr>
<tr>
<td>networkx.convert (module), 403</td>
</tr>
<tr>
<td>networkx.convert_matrix (module), 406</td>
</tr>
<tr>
<td>networkx.drawing.layout (module), 469</td>
</tr>
<tr>
<td>networkx.drawing.nx_agraph (module), 464</td>
</tr>
<tr>
<td>networkx.drawing.nx_pydot (module), 467</td>
</tr>
<tr>
<td>networkx.drawing.nx_pylab (module), 455</td>
</tr>
<tr>
<td>networkx.exception (module), 475</td>
</tr>
<tr>
<td>networkx.generators.atlas (module), 347</td>
</tr>
<tr>
<td>networkx.generators.bipartite (module), 379</td>
</tr>
<tr>
<td>networkx.generators.classic (module), 347</td>
</tr>
<tr>
<td>networkx.generators.degree_seq (module), 365</td>
</tr>
<tr>
<td>networkx.generators.directed (module), 372</td>
</tr>
<tr>
<td>networkx.generators.ego (module), 385</td>
</tr>
<tr>
<td>networkx.generators.geometric (module), 375</td>
</tr>
<tr>
<td>networkx.generators.hybrid (module), 379</td>
</tr>
<tr>
<td>networkx.generators.intersection (module), 386</td>
</tr>
<tr>
<td>networkx.generators.line (module), 383</td>
</tr>
<tr>
<td>networkx.generators.random_clustered (module), 371</td>
</tr>
<tr>
<td>networkx.generators.random_graphs (module), 356</td>
</tr>
<tr>
<td>networkx.generators.small (module), 352</td>
</tr>
<tr>
<td>networkx.generators.social (module), 388</td>
</tr>
<tr>
<td>networkx.generators.stochastic (module), 386</td>
</tr>
<tr>
<td>networkx.linalg.algebraicconnectivity (module), 394</td>
</tr>
<tr>
<td>networkx.linalg.attrmatrix (module), 397</td>
</tr>
<tr>
<td>networkx.linalg.graphmatrix (module), 389</td>
</tr>
<tr>
<td>networkx.linalg.laplacianmatrix (module), 391</td>
</tr>
<tr>
<td>networkx.linalg.spectrum (module), 393</td>
</tr>
<tr>
<td>networkx.readwrite.adjlist (module), 415</td>
</tr>
<tr>
<td>networkx.readwrite.edgelist (module), 422</td>
</tr>
<tr>
<td>networkx.readwrite.gexf (module), 429</td>
</tr>
<tr>
<td>networkx.readwrite.graphml (module), 431</td>
</tr>
<tr>
<td>networkx.readwrite.gpickle (module), 434</td>
</tr>
<tr>
<td>networkx.readwrite.graph6 (module), 444</td>
</tr>
<tr>
<td>networkx.readwrite.graphml (module), 435</td>
</tr>
<tr>
<td>networkx.readwrite.json_graph (module), 437</td>
</tr>
<tr>
<td>networkx.readwrite.leda (module), 442</td>
</tr>
<tr>
<td>networkx.readwrite.multiline_adjlist (module), 419</td>
</tr>
<tr>
<td>networkx.readwrite.nx_shp (module), 452</td>
</tr>
<tr>
<td>networkx.readwrite.nx_yaml (module), 443</td>
</tr>
<tr>
<td>networkx.readwrite.pajek (module), 450</td>
</tr>
<tr>
<td>networkx.readwrite.sparse6 (module), 447</td>
</tr>
<tr>
<td>networkx.utils (module), 477</td>
</tr>
<tr>
<td>networkx.utils.contextmanagers (module), 484</td>
</tr>
<tr>
<td>networkx.utils.decorators (module), 481</td>
</tr>
<tr>
<td>networkx.utils.misc (module), 477</td>
</tr>
<tr>
<td>networkx.utils.random_sequence (module), 478</td>
</tr>
<tr>
<td>networkx.utils.rcm (module), 482</td>
</tr>
<tr>
<td>networkx.utils.union_find (module), 478</td>
</tr>
<tr>
<td>NetworkXAlgorithmError (class in networkx), 475</td>
</tr>
<tr>
<td>NetworkXError (class in networkx), 475</td>
</tr>
<tr>
<td>NetworkXException (class in networkx), 475</td>
</tr>
<tr>
<td>NetworkXNoPath (class in networkx), 475</td>
</tr>
<tr>
<td>NetworkXPointlessConcept (class in networkx), 475</td>
</tr>
<tr>
<td>NetworkXUnbounded (class in networkx), 475</td>
</tr>
<tr>
<td>NetworkXUnfeasible (class in networkx), 475</td>
</tr>
<tr>
<td>newman_watts_strogatz_graph() (in module networkx.generators.random_graphs), 360</td>
</tr>
<tr>
<td>node, 493</td>
</tr>
<tr>
<td>node_boundary() (in module networkx.algorithms.boundary), 163</td>
</tr>
<tr>
<td>node_clique_number() (in module networkx.algorithms.clique), 189</td>
</tr>
<tr>
<td>node_connected_component() (in module networkx.algorithms.components.connected), 196</td>
</tr>
<tr>
<td>node_connectivity() (in module networkx.algorithms.connectivity.connectivity), 216</td>
</tr>
<tr>
<td>node_link_data() (in module networkx.readwrite.json_graph), 437</td>
</tr>
<tr>
<td>node_link_graph() (in module networkx.readwrite.json_graph), 438</td>
</tr>
<tr>
<td>node_redundancy() (in module networkx.algorithms.bipartite.redundancy), 157</td>
</tr>
<tr>
<td>nodes() (DiGraph method), 47</td>
</tr>
<tr>
<td>nodes() (Graph method), 20</td>
</tr>
<tr>
<td>nodes() (in module networkx.classes.function), 341</td>
</tr>
<tr>
<td>nodes() (MultiDiGraph method), 107</td>
</tr>
<tr>
<td>nodes() (MultiGraph method), 79</td>
</tr>
<tr>
<td>nodes_iter() (DiGraph method), 47</td>
</tr>
<tr>
<td>nodes_iter() (Graph method), 21</td>
</tr>
<tr>
<td>nodes_iter() (in module networkx.classes.function), 341</td>
</tr>
<tr>
<td>nodes_iter() (MultiDiGraph method), 107</td>
</tr>
<tr>
<td>nodes_iter() (MultiGraph method), 79</td>
</tr>
<tr>
<td>nodes_with_selfloops() (DiGraph method), 62</td>
</tr>
<tr>
<td>nodes_with_selfloops() (Graph method), 31</td>
</tr>
<tr>
<td>nodes_with_selfloops() (MultiDiGraph method), 122</td>
</tr>
<tr>
<td>nodes_with_selfloops() (MultiGraph method), 91</td>
</tr>
<tr>
<td>non_edges() (in module networkx.classes.function), 343</td>
</tr>
<tr>
<td>non_neighbors() (in module networkx.classes.function), 342</td>
</tr>
</tbody>
</table>
normalized_laplacian_matrix() (in module networkx.linalg.laplacianmatrix), 391
null_graph() (in module networkx.generators.classic), 351
number_attracting_components() (in module networkx.algorithms.components.attracting), 202
number_connected_components() (in module networkx.algorithms.components.connected), 195
number_of_cliques() (in module networkx.algorithms.clique), 189
number_of_edges() (DiGraph method), 61
number_of_edges() (Graph method), 31
number_of_edges() (in module networkx.classes.function), 343
number_of_edges() (MultiDiGraph method), 122
number_of_edges() (MultiGraph method), 90
number_of_nodes() (DiGraph method), 57
number_of_nodes() (Graph method), 29
number_of_nodes() (in module networkx.classes.function), 341
number_of_nodes() (MultiDiGraph method), 117
number_of_nodes() (MultiGraph method), 88
number_of_selfloops() (DiGraph method), 63
number_of_selfloops() (Graph method), 32
number_of_selfloops() (MultiDiGraph method), 123
number_of_selfloops() (MultiGraph method), 92
number_strongly_connected_components() (in module networkx.algorithms.components.strongly_connected), 197
number_weakly_connected_components() (in module networkx.algorithms.components.weakly_connected), 201
numeric_assortativity_coefficient() (in module networkx.algorithms.assortativity), 134
numerical_edge_match() (in module networkx.algorithms.isomorphism), 279
numerical_multiedge_match() (in module networkx.algorithms.isomorphism), 280
numerical_node_match() (in module networkx.algorithms.isomorphism), 278
out_degree() (MultiDiGraph method), 120
out_degree_centrality() (in module networkx.algorithms.centrality), 164
out_degree_iter() (DiGraph method), 60
out_degree_iter() (MultiDiGraph method), 110
out_edges() (DiGraph method), 49
out_edges() (MultiDiGraph method), 110
out_edges_iter() (DiGraph method), 50
out_edges_iter() (MultiDiGraph method), 110
overlap_weighted_projected_graph() (in module networkx.algorithms.bipartite.projection), 150

P
pagerank() (in module networkx.algorithms.link_analysis.pagerank_alg), 282
pagerank_numpy() (in module networkx.algorithms.link_analysis.pagerank_alg), 283
pagerank_scipy() (in module networkx.algorithms.link_analysis.pagerank_alg), 284
pappus_graph() (in module networkx.generators.small), 355
pareto_sequence() (in module networkx.utils.random_sequence), 479
parse_adjlist() (in module networkx.readwrite.adjlist), 417
parse_edgelist() (in module networkx.readwrite.edgelist), 428
parse_gml() (in module networkx.readwrite.gml), 433
parse_graph6() (in module networkx.readwrite.graph6), 445
parse_leda() (in module networkx.readwrite.leda), 443
parse_multiline_adjlist() (in module networkx.readwrite.multiline_adjlist), 421
parse_pajek() (in module networkx.readwrite.pajek), 451
parse_sparse6() (in module networkx.readwrite.sparse6), 448
path_graph() (in module networkx.generators.classic), 351
periphery() (in module networkx.algorithms.distance_measures), 235
petersen_graph() (in module networkx.algorithms.distance_measures), 235
powerlaw_cluster_graph() (in module networkx.generators.random_graphs), 363
powerlaw_sequence() (in module networkx.utils.random_sequence), 479
predecessor() (in module networkx.algorithms.shortest_paths.unweighted), 314
predecessors() (DiGraph method), 53
predecessors() (MultiDiGraph method), 114
predecessors_iter() (DiGraph method), 53
predecessors_iter() (MultiDiGraph method), 114
preferential_attachment() (in module networkx.algorithms.link_prediction), 292
preflow_push() (in module networkx.algorithms.flow), 252
projected_graph() (in module networkx.algorithms.bipartite.projection), 147
pydot_layout() (in module networkx.drawing.nx_pydot), 469
pygraphviz_layout() (in module networkx.drawing.nx_agraph), 467
ra_index_soundarajan_hopcroft() (in module networkx.algorithms.link_prediction), 294
radius() (in module networkx.algorithms.distance_measures), 235
ramsey_R2() (in module networkx.algorithms.approximation.ramsey), 131
random_clustered_graph() (in module networkx.generators.random_clustered), 371
random_degree_sequence_graph() (in module networkx.generators.degree_seq), 370
random_geometric_graph() (in module networkx.generators.geometric), 375
random_layout() (in module networkx.drawing.layout), 470
random_lobster() (in module networkx.generators.random_graphs), 363
random_powerlaw_tree() (in module networkx.generators.directed), 364
random_powerlaw_tree_sequence() (in module networkx.generators.random_graphs), 365
random_regular_graph() (in module networkx.generators.random_graphs), 362
random_shell_graph() (in module networkx.generators.random_graphs), 364
random_weighted_sample() (in module networkx.utils.random_sequence), 481
read_adjlist() (in module networkx.readwrite.adjlist), 415
read_dot() (in module networkx.drawing.nx_agraph), 466
read_dot() (in module networkx.drawing.nx_pydot), 469
read_edgelist() (in module networkx.readwrite.edgelist), 423
read_gexf() (in module networkx.readwrite.gexf), 429
read_gml() (in module networkx.readwrite.gml), 434
read_gpickle() (in module networkx.readwrite.gpickle), 431
read_graph6() (in module networkx.readwrite.graph6), 445
read_graphml() (in module networkx.readwrite.graphml), 436
read_leda() (in module networkx.readwrite.leda), 442
read_multiline_adjlist() (in module networkx.readwrite.multiline_adjlist), 419
read_pajek() (in module networkx.readwrite.pajek), 451
read_shp() (in module networkx.readwrite.nx_shp), 452
read_sparse6() (in module networkx.readwrite.sparse6), 448
read_weighted_edgelist() (in module networkx.readwrite.edgelist), 425
read_yaml() (in module networkx.readwrite.nx_yaml), 443
relabel_gexf_graph() (in module networkx.readwrite.gexf), 431
remove_edge() (DiGraph method), 43
remove_edge() (Graph method), 17
remove_edge() (MultiDiGraph method), 103
remove_edge() (MultiGraph method), 74
remove_edges_from() (DiGraph method), 44
remove_edges_from() (Graph method), 18
remove_edges_from() (MultiDiGraph method), 104
remove_edges_from() (MultiGraph method), 75
remove_node() (DiGraph method), 40
remove_node() (Graph method), 14
remove_node() (MultiDiGraph method), 100
remove_node() (MultiGraph method), 71
remove_nodes_from() (DiGraph method), 41
remove_nodes_from() (Graph method), 14
remove_nodes_from() (MultiDiGraph method), 100
remove_nodes_from() (MultiGraph method), 71
resource_allocation_index() (in module networkx.algorithms.link_prediction), 290
reverse() (DiGraph method), 66
reverse() (in module networkx.algorithms.operators.unary), 300
reverse() (MultiDiGraph method), 126
reverse_cuthill_mckee_ordering() (in module networkx.utils.rcm), 483
reversed() (in module networkx.utils.contextmanagers), 485
rich_club_coefficient() (in module networkx.algorithms.richclub), 307
robinson_alexander_clustering() (in module networkx.algorithms.bipartite.cluster), 157
scale_free_graph() (in module networkx.generators.directed), 374
sedgewick_maze_graph() (in module networkx.generators.small), 355
selfloop_edges() (DiGraph method), 62
selfloop_edges() (Graph method), 32
selfloop_edges() (MultiDiGraph method), 123
semantic_feasibility() (DiGraphMatcher method), 276
semantic_feasibility() (GraphMatcher method), 275
set_edge_attributes() (in module networkx.classes.function), 344
set_node_attributes() (in module networkx.classes.function), 343
sets() (in module networkx.algorithms.bipartite.basic), 144
shell_layout() (in module networkx.drawing.layout), 471
shortest_augmenting_path() (in module networkx.algorithms.flow), 250
shortest_path() (in module networkx.algorithms.shortest_paths.generic), 308
shortest_path_length() (in module networkx.algorithms.shortest_paths.generic), 310
simple_cycles() (in module networkx.algorithms.cycles), 230
single_source_dijkstra() (in module networkx.algorithms.shortest_paths.weighted), 319
single_source_dijkstra_path() (in module networkx.algorithms.shortest_paths.weighted), 316
single_source_dijkstra_path_length() (in module networkx.algorithms.shortest_paths.weighted), 317
single_source_shortest_path() (in module networkx.algorithms.shortest_paths.unweighted), 312
single_source_shortest_path_length() (in module networkx.algorithms.shortest_paths.unweighted), 313
size() (DiGraph method), 61
size() (Graph method), 30
size() (MultiDiGraph method), 121
size() (MultiGraph method), 89
spectral_bipartivity() (in module networkx.algorithms.bipartite.spectral), 152
spectral_layout() (in module networkx.drawing.layout), 472
spectralordering() (in module networkx.linalg.algebraicconnectivity), 396
spring_layout() (in module networkx.drawing.layout), 471
square_clustering() (in module networkx.algorithms.cluster), 193
star_graph() (in module networkx.generators.classic), 351
stochastic_graph() (in module networkx.generators.stochastic), 386
stoer_wagner() (in module networkx.algorithms.connectivity.stoerwagner), 223
strong_product() (in module networkx.algorithms.operators.product), 306
strongly_connected_component_subgraphs() (in module networkx.algorithms.components.strongly_connected), 198
strongly_connected_components() (in module networkx.algorithms.components.strongly_connected), 198
strongly_connected_components_recursive() (in module networkx.algorithms.components.strongly_connected), 199
subgraph() (DiGraph method), 65
subgraph() (Graph method), 35
subgraph() (MultiDiGraph method), 126
subgraph() (MultiGraph method), 94
subgraph_is_isomorphic() (DiGraphMatcher method), 276
subgraph_is_isomorphic() (GraphMatcher method), 274
subgraph_isomorphisms_iter() (DiGraphMatcher method), 276
subgraph_isomorphisms_iter() (GraphMatcher method), 274
successors() (DiGraph method), 53
successors() (MultiDiGraph method), 113
successors_iter() (DiGraph method), 53
successors_iter() (MultiDiGraph method), 113
symmetric_difference() (in module networkx.algorithms.operators.binary), 303
syntactic_feasibility() (DiGraphMatcher method), 277
syntactic_feasibility() (GraphMatcher method), 275
tensor_product() (in module networkx.algorithms.operators.product), 306
tetrahedral_graph() (in module networkx.generators.small), 355
to_agraph() (in module networkx.drawing.nx_agraph), 465
to_dict_of_dicts() (in module networkx.convert), 404
to_dict_of_lists() (in module networkx.convert), 405
to_directed() (DiGraph method), 64
to_directed() (Graph method), 34
to_directed() (MultiDiGraph method), 125
to_directed() (MultiGraph method), 93
to_edgelist() (in module networkx.convert), 406
to_networkx_graph() (in module networkx.convert), 403
to_numpy_matrix() (in module networkx.convert_matrix), 408
to_numpy_recarray() (in module networkx.convert_matrix), 409
to_pydot() (in module networkx.drawing.nx_pydot), 468
to_scipy_sparse_matrix() (in module networkx.convert_matrix), 411
to_undirected() (DiGraph method), 64

to_undirected() (Graph method), 33

to_undirected() (MultiDiGraph method), 124

to_undirected() (MultiGraph method), 93

topological_sort() (in module networkx.algorithms.dag), 232

topological_sort_recursive() (in module networkx.algorithms.dag), 232

transitivity() (in module networkx.algorithms.cluster), 190

tree_data() (in module networkx.readwrite.json_graph), 441

tree_graph() (in module networkx.readwrite.json_graph), 441

triangles() (in module networkx.algorithms.cluster), 190

trivial_graph() (in module networkx.generators.classic), 351

truncated_cube_graph() (in module networkx.generators.small), 355

truncated_tetrahedron_graph() (in module networkx.generators.small), 356

tutte_graph() (in module networkx.generators.small), 356

U

uniform_random_intersection_graph() (in module networkx.generators.intersection), 386

uniform_sequence() (in module networkx.utils.random_sequence), 479

union() (in module networkx.algorithms.operators.binary), 301

union() (UnionFind method), 478

union_all() (in module networkx.algorithms.operators.all), 303

W

watts_strogatz_graph() (in module networkx.generators.random_graphs), 360

waxman_graph() (in module networkx.generators.geometric), 377

weakly_connected_component_subgraphs() (in module networkx.algorithms.components.weakly_connected), 201

weakly_connected_components() (in module networkx.algorithms.components.weakly_connected), 201

weighted_choice() (in module networkx.utils.random_sequence), 481

weighted_projected_graph() (in module networkx.algorithms.bipartite.projection), 148

wheel_graph() (in module networkx.generators.classic), 352

within_inter_cluster() (in module networkx.algorithms.link_prediction), 295

write_adjlist() (in module networkx.readwrite.adjlist), 417

write_dot() (in module networkx.drawing.nx_agraph), 466

write_dot() (in module networkx.drawing.nx_pydot), 468

write_edgelist() (in module networkx.readwrite.edgelist), 424

write_gexf() (in module networkx.readwrite.gexf), 430

write_gml() (in module networkx.readwrite.gml), 432

write_gpickle() (in module networkx.readwrite.gpickle), 435

write_graph6() (in module networkx.readwrite.graph6), 447

write_graphml() (in module networkx.readwrite.graphml), 436

write_multiline_adjlist() (in module networkx.readwrite.multiline_adjlist), 420

write_pajek() (in module networkx.readwrite.pajek), 451

write_shp() (in module networkx.readwrite.nx_shp), 452

write_sparse6() (in module networkx.readwrite.sparse6), 450

write_weighted_edgelist() (in module networkx.readwrite.edgelist), 426

write_yaml() (in module networkx.readwrite.nx_yaml), 444

Z

zipf_rv() (in module networkx.utils.random_sequence), 480

zipf_sequence() (in module networkx.utils.random_sequence), 480