

networkx

API Documentation

August 21, 2005

Contents

Contents	1
1 Package networkx	5
1.1 Modules	5
1.2 Variables	7
2 Module networkx.base	8
2.1 Functions	12
2.2 Variables	13
2.3 Class DiGraph	13
2.3.1 Methods	15
2.4 Class Graph	21
2.4.1 Methods	21
2.5 Class NetworkXError	29
2.5.1 Methods	29
2.6 Class NetworkXException	29
2.6.1 Methods	29
3 Module networkx centrality	30
3.1 Functions	30
3.2 Variables	31
4 Module networkx cliques	32
4.1 Functions	32
4.2 Variables	35
5 Module networkx cluster	36
5.1 Functions	36
5.2 Variables	36
6 Module networkx cores	38
6.1 Functions	38
6.2 Variables	38

7 Package networkx.drawing	39
7.1 Modules	39
8 Module networkx.drawing.layout	40
8.1 Functions	40
8.2 Variables	40
9 Module networkx.drawing.nx_pydot	42
9.1 Functions	42
9.2 Variables	42
10 Module networkx.drawing.nx_pylab	44
10.1 Functions	44
10.2 Variables	46
11 Module networkx.drawing.nx_vtk	47
11.1 Functions	47
11.2 Variables	47
12 Package networkx.generators	48
12.1 Modules	48
13 Module networkx.generators.atlas	49
13.1 Functions	49
13.2 Variables	49
14 Module networkx.generators.classic	50
14.1 Functions	50
14.2 Variables	54
15 Module networkx.generators.degree_seq	55
15.1 Functions	55
15.2 Variables	58
16 Module networkx.generators.geometric	59
16.1 Functions	59
16.2 Variables	59
17 Module networkx.generators.random_graphs	60
17.1 Functions	60
17.2 Variables	65
18 Module networkx.generators.small	66
18.1 Functions	66
18.2 Variables	70
19 Module networkx.hybrid	71
19.1 Functions	71
19.2 Variables	71

20 Module networkx.io	72
20.1 Functions	72
20.2 Variables	74
21 Module networkx.isomorph	75
21.1 Functions	75
21.2 Variables	75
22 Module networkx.operators	76
22.1 Functions	76
22.2 Variables	79
23 Module networkx.paths	80
23.1 Functions	80
23.2 Variables	82
24 Module networkx.queues	84
24.1 Variables	84
24.2 Class BFS	84
24.2.1 Methods	84
24.3 Class DFS	85
24.3.1 Methods	85
24.4 Class FIFO	85
24.4.1 Methods	85
24.5 Class LIFO	86
24.5.1 Methods	86
24.6 Class Priority	86
24.6.1 Methods	86
24.7 Class Random	87
24.7.1 Methods	87
24.8 Class RFS	87
24.8.1 Methods	87
25 Module networkx.release	88
25.1 Variables	88
26 Module networkx.search	89
26.1 Functions	89
26.2 Variables	91
27 Module networkx.search_class	92
27.1 Variables	92
27.2 Class Forest	92
27.2.1 Methods	93
27.3 Class Length	93
27.3.1 Methods	93
27.4 Class Postorder	94
27.4.1 Methods	94
27.5 Class Predecessor	95

27.5.1	Methods	95
27.6	Class Preorder	95
27.6.1	Methods	96
27.7	Class Search	96
27.7.1	Methods	97
27.8	Class Successor	98
27.8.1	Methods	98
28	Module <code>networkx.spectrum</code>	100
28.1	Functions	100
28.2	Variables	100
29	Module <code>networkx.threshold</code>	101
29.1	Functions	101
29.2	Variables	106
30	Module <code>networkx.utils</code>	108
30.1	Functions	108
30.2	Variables	109
31	Module <code>networkx.xbase</code>	110
31.1	Variables	114
31.2	Class <code>XDiGraph</code>	115
31.2.1	Methods	117
31.3	Class <code>XGraph</code>	125
31.3.1	Methods	126
	Index	134

1 Package networkx

NetworkX

NetworkX (NX) is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks.

Using

Just write in Python

```
>>> import networkx as NX
>>> G=NX.Graph()
>>> G.add_edge(1,2)
>>> G.add_node("spam")
>>> print G.nodes()
[1, 2, 'spam']
>>> print G.edges()
[(1, 2)]
```

See `networkx.base` for the details of the API.

1.1 Modules

- **base:**
Base classes for graphs and digraphs.
(Section 2, p. 8)
- **centrality:**
Centrality measures.
(Section 3, p. 30)
- **cliques:**
Cliques - Find and manipulate cliques of graphs
(Section 4, p. 32)
- **cluster:**
Compute clustering coefficients and transitivity of graphs.
(Section 5, p. 36)
- **cores:**
Find and manipulate the k-cores of a graph
(Section 6, p. 38)
- **drawing** (Section 7, p. 39)
 - **layout:**
Layout (positioning) algorithms for graph drawing.
(Section 8, p. 40)
 - **nx_pydot:**
Import and export networkx networks to dot format using pydot.
(Section 9, p. 42)
 - **nx_pylab:**
Draw networks with matplotlib/pylab.

- (Section 10, p. 44)*
 - **nx_vtk:**
Draw networks in 3d with vtk.
(Section 11, p. 47)
- **generators:**
A package for generating various graphs in networkx.
(Section 12, p. 48)
 - **atlas:**
Generators for the small graph atlas.
(Section 13, p. 49)
 - **classic:**
Generators for some classic graphs.
(Section 14, p. 50)
 - **degree_seq:**
Generate graphs with a given degree sequence.
(Section 15, p. 55)
 - **geometric:**
Generators for geometric graphs.
(Section 16, p. 59)
 - **random_graphs:**
Generators for random graphs
(Section 17, p. 60)
 - **small:**
Various small and named graphs, together with some compact generators.
(Section 18, p. 66)
- **hybrid:**
Hybrid
(Section 19, p. 71)
- **io:**
Read and write graphs and networks.
(Section 20, p. 72)
- **isomorph:**
Fast checking to see if graphs are not isomorphic.
(Section 21, p. 75)
- **operators:**
Operations on graphs; including union, complement, subgraph.
(Section 22, p. 76)
- **paths:**
Shortest paths, diameter, radius, eccentricity, and related methods.
(Section 23, p. 80)
- **queues:**
Helper queues for use in graph searching.
(Section 24, p. 84)
- **release:**
Release data for NetworkX.
(Section 25, p. 88)
- **search:**
Search algorithms, shortest path, spanning trees, etc.

(Section 26, p. 89)

- **search_class:**

Graph search classes

(Section 27, p. 92)

- **spectrum:**

Laplacian, adjacency matrix, and spectrum of graphs.

(Section 28, p. 100)

- **threshold:**

Threshold Graphs - Creation, manipulation and identification.

(Section 29, p. 101)

- **utils:**

Utilities for networkx package

(Section 30, p. 108)

- **xbase:**

Methods for general graphs (XGraph) and digraphs (XDiGraph)

allowing self-loops, multiple edges, arbitrary (hashable) objects as nodes and arbitrary objects associated with edges.

(Section 31, p. 110)

1.2 Variables

Name	Description
<code>__author__</code>	Value: 'Aric Hagberg <hagberg@lanl.gov>\nDan Schult <-dschult@colgate.edu>\nPieter Sw... (<i>type=str</i>)
<code>__date__</code>	Value: 'Sun Aug 21 08:07:12 2005' (<i>type=str</i>)
<code>__license__</code>	Value: 'LGPL' (<i>type=str</i>)
<code>__version__</code>	Value: '0.24' (<i>type=str</i>)

2 Module `networkx.base`

Base classes for graphs and digraphs.

Unless otherwise specified, by graph we mean a simple graph that has no self-loops or multiple (parallel) edges. (See the module `xbase.py` for graph classes `XGraph` and `XDiGraph` that allow for self-loops, multiple edges and arbitrary objects associated with edges.)

The following classes are provided:

`Graph`

The basic operations common to graph-like classes.

`DiGraph`

Operations common to digraphs, a graph with directed edges.
Subclass of `Graph`.

An empty graph or digraph is created with

```
G=Graph()
```

```
G=DiGraph()
```

This module implements graphs using data structures based on an adjacency list implemented as a node-centric dictionary of dictionaries. The dictionary contains keys corresponding to the nodes and the values are dictionaries of neighboring node keys with the value 1. This allows fast addition, deletion and lookup of nodes and neighbors in large graphs. The underlying datastructure should only be visible in this module. In all other modules, instances of graph-like objects are manipulated solely via the methods defined here and not by acting directly on the datastructure.

The following notation is used throughout NetworkX documentation and code: (we use mathematical notation n, v, w, \dots to indicate a node, $v=\text{vertex}=\text{node}$).

$G, G_1, G_2, H, \text{etc.}$:

Graphs

$n, n_1, n_2, u, v, v_1, v_2$:

nodes (v stands for $\text{vertex}=\text{node}$)

`nlist, vlist:`

a list of nodes

`nbunch:`

a "bunch" of nodes (vertices). an nbunch is any iterable container of nodes that is not itself a node in the graph. (It can be an iterable or an iterator, e.g. a list, set, graph, file, etc..)

`e=(n1,n2):`

an edge (a python "2-tuple"), also written `u-v`. In Xgraph `G.add_edge(n1,n2)` is equivalent to `add_edge(n1,n2,1)`. However, `G.delete_edge(n1,n2)` will delete all edges between `n1` and `n2`.

`elist:`

a list of edges (as tuples)

`ebunch:`

a bunch of edges (as tuples)

an ebunch is any iterable (non-string) container of edge-tuples. (Similar to nbunch, also see `add_edge`).

Warning:

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

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

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

Methods

The Graph class provides rudimentary graph operations:

Mutating Graph methods

`G.add_node(n)`, `G.add_nodes_from(nbunch)`

`G.delete_node(n)`, `G.delete_nodes_from(nbunch)`

`G.add_edge(n1,n2)`, `G.add_edge(e)`, where `e=(u,v)`

`G.add_edges_from(ebunch)`

`G.delete_edge(n1,n2)`, `G.delete_edge(e)`, where `e=(u,v)`

`G.delete_edges_from(ebunch)`

`G.add_path(nlist)`

`G.add_cycle(nlist)`

`G.clear()`

`G.subgraph(nbunch,inplace=True)`

Non-mutating Graph methods

`len(G)`

`n in G` (equivalent to `G.has_node(n)`)

`G.has_node(n)`

`G.nodes()`

`G.nodes_iter()`

`G.has_edge(n1,n2)`

`G.edges()`, `G.edges(n)`, `G.edges(nbunch)`

`G.edges_iter()`, `G.edges_iter(n)`, `G.edges_iter(nbunch)`

`G.neighbors(n)`

`G[n]` (equivalent to `G.neighbors(n)`)

`G.neighbors_iter(n)` # iterator over neighbors

`G.number_of_nodes()`

`G.number_of_edges()`

`G.degree(n)`, `G.degree(nbunch)`

`G.degree_iter(n)`, `G.degree_iter(nbunch)`

`G.is_directed()`

Methods returning a new graph

`G.subgraph(nbunch)`

`G.subgraph(nbunch,create_using=H)`

`G.copy()`

`G.to_undirected()`

`G.to_directed()`

Examples

Create an empty graph structure (a "null graph") with zero nodes and zero edges.

```
>>> from networkx import *
>>> G=Graph()
```

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

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

by adding a list of nodes:

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

by using an iterator:

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

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

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

(H can be another graph, or dict, or set, or even a file.)

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

(Any hashable object can represent a node, e.g. a Graph,
a customized node object, etc.)

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

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

by adding a list of edges:

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

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

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

There are no complaints when adding existing nodes or edges:

```
>>> G=Graph()
>>> G.add_edge([(1,2),(1,3)])
```

will add new nodes as required.

2.1 Functions

degree (<i>G</i> , <i>nbunch</i> =None, <i>with_labels</i> =False)

Return degree of single node or of nbunch of nodes. If nbunch is omitted, then return degrees of all nodes.

degree_histogram (<i>G</i>)

Return a list of the frequency of each degree value.

The degree values are the index in the list. Note: the bins are width one, hence len(list) can be large (Order(number_of_edges))

density (<i>G</i>)

Return the density of a graph.

density = size/(order*(order-1)/2) density()=0.0 for an edge-less graph and 1.0 for a complete graph.

edges (<i>G</i> , <i>nbunch</i> =None, <i>with_labels</i> =False)

Return list of edges adjacent to nodes in nbunch. Return all edges if nbunch is unspecified or nbunch=None.

edges_iter (<i>G</i> , <i>nbunch</i> =None, <i>with_labels</i> =False)

Return iterator over edges adjacent to nodes in nbunch. Return all edges if nbunch is unspecified or nbunch=None.

<code>neighbors(<i>G</i>, <i>n</i>, <i>with_labels</i>=False)</code>

Return a list of nodes connected to node <i>n</i> .

<code>nodes(<i>G</i>)</code>

Return a copy of the graph nodes in a list.

<code>nodes_iter(<i>G</i>)</code>

Return an iterator over the graph nodes.

<code>number_of_edges(<i>G</i>)</code>

Return the size of a graph = number of edges.

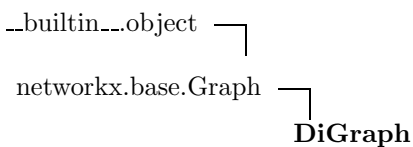
<code>number_of_nodes(<i>G</i>)</code>

Return the order of a graph = number of nodes.

2.2 Variables

Name	Description
<code>__author__</code>	Value: 'Aric Hagberg (hagberg@lanl.gov)\nPieter Swart - (swart@lanl.gov)\nDan Schult(d... (<i>type=str</i>)
<code>__credits__</code>	Value: '' (<i>type=str</i>)
<code>__date__</code>	Value: '\$Date: 2005-06-24 14:16:40 -0600 (Fri, 24 Jun - 2005) \$\n' (<i>type=str</i>)
<code>__revision__</code>	Value: '\$Revision: 1061 \$\n' (<i>type=str</i>)

2.3 Class `DiGraph`



Known Subclasses: `XDigraph`

A graph with directed edges. Subclass of `Graph`.

`DiGraph` inherits from `Graph`, overriding the following methods:

`__init__`: replaces `self.adj` with the dicts `self.pred` and `self.succ`

`--getitem--``add_node``delete_node``add_edge``delete_edge``add_nodes_from``delete_nodes_from``add_edges_from``delete_edges_from``edges_iter``degree_iter``degree``copy``clear``subgraph``is_directed``to_directed``to_undirected`

Digraph adds the following methods to those of Graph:

`successors``successors_iter``predecessors``predecessors_iter``out_degree`

`out_degree_iter`

`in_degree`

`in_degree_iter`

2.3.1 Methods

`__init__(self, **kwargs)`

Initialize Graph.

`>>> G=Graph(name="empty")` creates empty graph G with `G.name="empty"`

Overrides: `networkx.base.Graph.__init__` `exitit`(inherited documentation)

`__getitem__(self, n)`

Return the in- and out-neighbors of node `n` as a list.

This provides digraph `G` the natural property that `G[n]` returns the neighbors of `G`.

Overrides: `networkx.base.Graph.__getitem__`

`add_edge(self, u, v=None)`

Add a single directed edge `(u,v)` to the digraph.

Can be used in two basic forms:

`G.add_edge(u,v)` or `G.add_edge((u,v))` are equivalent forms of adding a single edge between nodes `u` and `v`. Nodes are not required to exist before adding an edge; they will be added in silence.

For example, the following examples all add the edge `(1,2)` to the digraph `G`.

```
>>> G=DiGraph()
>>> G.add_edge( 1, 2 )           # explicit two node form
>>> G.add_edge( (1,2) )         # single edge as tuple of two nodes
>>> G.add_edges_from( [(1,2)] ) # list of edges form
```

Overrides: `networkx.base.Graph.add_edge`

`add_edges_from(self, ebunch)`

Add all the edges in ebunch to the graph.

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

See `add_edge` for an example.

Overrides: `networkx.base.Graph.add_edges_from`

`add_node(self, n)`

Add a single node to the digraph.

n can be any hashable object (it is used as a key in a dictionary). On many platforms this includes mutables such as Graphs e.g., though one should be careful the hash doesn't change during the lifetime of the graph.

```
>>> G=DiGraph()
>>> K3=complete_graph(3)
>>> G.add_nodes_from(K3)      # add the nodes from K3 to G
>>> G.nodes()
[1,2,3]
>>> G.clear()
>>> G.add_node(K3)            # add the graph K3 as a node in G.
>>> number_of_nodes(G)
1
```

Overrides: `networkx.base.Graph.add_node`

`add_nodes_from(self, nbunch)`

Add multiple nodes to the digraph.

nbunch:

A container of nodes that will be iterated through once (thus it can be an iterator or an iterable). A node can be any hashable object (it is used as a key in a dictionary)

Overrides: `networkx.base.Graph.add_nodes_from`

`clear(self)`

Remove name and delete all nodes and edges from digraph.

Overrides: `networkx.base.Graph.clear`

copy(*self*)

Return a (shallow) copy of the digraph.

Identical to `dict.copy()` of adjacency dicts `pred` and `succ`, with `name` and `dna` copied as well.

Overrides: `networkx.base.Graph.copy`

degree(*self*, *nbunch*=None, *with_labels*=False)

Return degree of single node or of *nbunch* of nodes.

If *nbunch* is omitted or *nbunch*=None, then return degrees of all nodes.

If *nbunch* is a single node *n*, return degree of *n*.

If *nbunch* is an iterable (non-string) container of nodes, return a list of values, one for each *n* in *nbunch*. (omitting *nbunch* or *nbunch*=None is interpreted as *nbunch* = all nodes in graph.)

If *with_labels*==True, then return a dict that maps each *n* in *nbunch* to `degree(n)`.

Any nodes in *nbunch* that are not in the graph are (quietly) ignored.

Overrides: `networkx.base.Graph.degree`

degree_iter(*self*, *nbunch*=None, *with_labels*=False)

Return iterator that return `degree(n)` or `(n,degree(n))` for all *n* in *nbunch*. If *nbunch* is omitted, then iterate over all nodes.

nbunch: a singleton node, a string (which is treated

as a singleton node), or any iterable (non-string) container of nodes for which `len(nbunch)` is defined. For example, a list, dict, set, Graph, numeric array, or user-defined iterable object.

If *with_labels*=True, iterator will return an `(n,degree(n))` tuple of node and degree.

Any nodes in *nbunch* but not in the graph will be (quietly) ignored.

Overrides: `networkx.base.Graph.degree_iter`

`delete_edge(self, u, v=None)`

Delete the single directed edge (u,v) from the digraph.

Can be used in two basic forms: `G.delete_edge(u,v)` or `G.delete_edge((u,v))` are equivalent forms of deleting a directed edge `u->v`. If the nodes do not exist; return without complaining.

Overrides: `networkx.base.Graph.delete_edge`

`delete_edges_from(self, ebunch)`

Delete the directed edges in ebunch from the digraph.

ebunch: Container of 2-tuples (u,v). The container must be iterable or an iterator. It is iterated over once. Edges that are not in the digraph are ignored.

Overrides: `networkx.base.Graph.delete_edges_from`

`delete_node(self, n)`

Delete node n from the digraph.

Attempting to delete a non-existent node will raise a `NetworkXError`.

Overrides: `networkx.base.Graph.delete_node`

`delete_nodes_from(self, nbunch)`

Remove nodes in nbunch from the digraph.

nbunch: an iterable or iterator containing valid (hashable) node names.

Attempting to delete a non-existent node will raise an exception. This could mean some nodes in nbunch were deleted and some valid nodes were not!

Overrides: `networkx.base.Graph.delete_nodes_from`

`edges_iter(self, nbunch=None, with_labels=False)`

Return iterator that iterates once over each edge adjacent to nodes in `nbunch`, or over all edges in digraph if no nodes are specified.

See `add_node` for definition of `nbunch`.

Those nodes in `nbunch` that are not in the graph will be (quietly) ignored.

`with_labels=True` is not supported (in that case you should probably use `neighbors()`)

Overrides: `networkx.base.Graph.edges_iter`

`foo(self)`

`in_degree(self, nbunch=None, with_labels=False)`

Return in-degree of single node or of `nbunch` of nodes. If `nbunch` is omitted or `nbunch=None`, then return in-degrees of all nodes.

`in_degree_iter(self, nbunch=None, with_labels=False)`

Return iterator that return `in_degree(n)` or `(n,in_degree(n))` for all `n` in `nbunch`. If `nbunch` is ommitted, then iterate over all nodes.

See `degree_iter` method for Digraph Class for more details.

`is_directed(self)`

Return True if a directed graph.

Overrides: `networkx.base.Graph.is_directed`

`neighbors(self, n, with_labels=False)`

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

If `with_labels=True`, return a dict keyed by neighbors.

Overrides: `networkx.base.Graph.neighbors`

`neighbors_iter(self, n, with_labels=False)`

Return an iterator over all nodes connected to node `n`.

If `with_labels=True`, return an iterator of `(neighbor, 1)` tuples.

Overrides: `networkx.base.Graph.neighbors_iter`

`out_degree(self, nbunch=None, with_labels=False)`

Return out-degree of single node or of `nbunch` of nodes.

If `nbunch` is omitted or `nbunch=None`, then return out-degrees of all nodes.

`out_degree_iter(self, nbunch=None, with_labels=False)`

Return iterator that return `out_degree(n)` or `(n, out_degree(n))` for all `n` in `nbunch`. If `nbunch` is omitted, then iterate over all nodes.

See `degree_iter` method for `Digraph` Class for more details.

`predecessors(self, v, with_labels=False)`

Return predecessor nodes of `v`.

`predecessors_iter(self, v, with_labels=False)`

Return an iterator for predecessor nodes of `v`.

`reverse(self)`

Return a new digraph with the same vertices and edges as `G` but with the directions of the edges reversed.

`successors(self, v, with_labels=False)`

Return sucessor nodes of `v`.

`successors_iter(self, v, with_labels=False)`

Return an iterator for successor nodes of `v`.

`to_directed(self)`

Return a directed representation of the digraph.

This is already directed, so merely return a copy.

Overrides: `networkx.base.Graph.to_directed`

to_undirected(*self*)

Return the undirected representation of the digraph.

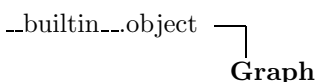
A new graph is returned (the underlying graph). The edge u-v is in the underlying graph if either u->v or v->u is in the digraph.

Overrides: networkx.base.Graph.to_undirected

Inherited from object: __delattr__, __getattr__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__

Inherited from Graph: __contains__, __iter__, __len__, __str__, add_cycle, add_path, edge_boundary, edges, has_edge, has_neighbor, has_node, node_boundary, nodes, nodes_iter, number_of_edges, number_of_nodes, order, print_dna, size, subgraph

2.4 Class Graph



Known Subclasses: DiGraph, XGraph

Graph is a simple graph without any multiple (parallel) edges or self-loops. Attempting to add either will not change the graph and will not report an error.

2.4.1 Methods

__init__(*self*, *kws*)**

Initialize Graph.

```
>>> G=Graph(name="empty") creates empty graph G with G.name="empty"
```

Overrides: __builtin__.object.__init__

__contains__(*self*, *n*)

Return True if *n* is a node in graph.

Allows the expression '*n* in *G*'.

Testing whether an unhashable object, such as a list, is in the dict datastructure (self.adj) will raise a TypeError. Rather than propagate this to the calling method, just return False.

`__getitem__(self, n)`

Return the neighbors of node `n` as a list.

This provides graph `G` the natural property that `G[n]` returns the neighbors of `G`.

`__iter__(self)`

Return an iterator over the nodes in `G`.

This is the iterator for the underlying adjacency dict.
(Allows the expression 'for `n` in `G`')

`__len__(self)`

Return the number of nodes in graph.

`__str__(self)`

Overrides: `__builtin__.object.__str__`

`add_cycle(self, nlist)`

Add the cycle of nodes in `nlist` to graph

`add_edge(self, u, v=None)`

Add a single edge (`u,v`) to the graph.

Can be used in two basic forms:
`G.add_edge(u,v)` or `G.add_edge((u,v))` are equivalent forms of adding a single edge between nodes `u` and `v`. Nodes are not required to exist before adding an edge; they will be added in silence.

The following examples all add the edge (1,2) to graph `G`.

```
>>> G=Graph()
>>> G.add_edge( 1, 2 )           # explicit two node form
>>> G.add_edge( (1,2) )         # single edge as tuple of two nodes
>>> G.add_edges_from( [(1,2)] ) # add edges from iterable container
```

`add_edges_from(self, ebunch)`

Add all the edges in ebunch to the graph.

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

`add_node(self, n)`

Add a single node n to the graph.

The node n can be any hashable object (it is used as a key in a dictionary). On many platforms this includes mutables such as Graphs e.g., though one should be careful the hash doesn't change on mutables.

Examples:

```
>>> G=Graph()
>>> K3=complete_graph(3)
>>> G.add_node(1)
>>> G.add_node('Hello')
>>> G.add_node(K3)
>>> G.nodes()
[1, 'Hello', <networkx.base.Graph object at 0x5f7430>]
```

`add_nodes_from(self, nbunch)`

Add multiple nodes to the graph.

nbunch:

A container of nodes that will be iterated through once (thus it can be an iterator or an iterable) Each element of the container should be hashable.

Examples:

```
>>> G=Graph()
>>> K3=complete_graph(3)
>>> G.add_nodes_from(1)
NetworkXError, Container 1 is not an iterator or iterable. Use add_node?
>>> G.add_nodes_from('Hello')
>>> G.add_nodes_from(K3)
>>> G.nodes()
['H', 'e', 'l', 'l', 'o', 1, 2, 3]
```

`add_path(self, nlist)`

Add the path through the nodes in `nlist` to graph

`clear(self)`

Remove name and delete all nodes and edges from graph.

`copy(self)`

Return a (shallow) copy of the graph.

Identical to `dict.copy()` of adjacency dict `adj`, with name and `dna` copied as well.

`degree(self, nbunch=None, with_labels=False)`

Return degree of single node or of `nbunch` of nodes.

If `nbunch` is omitted or `nbunch=None`, then return degrees of all nodes.

The degree of a node is the number of edges attached to that node.

Can be called in three ways:

```
G.degree(n):          return the degree of node n
G.degree(nbunch):     return a list of values, one for each n in nbunch
(nbunch is any iterable container of nodes.)
G.degree():           same as nbunch = all nodes in graph.
```

If `with_labels==True`, then return a dict that maps each `n` in `nbunch` to `degree(n)`.

Any nodes in `nbunch` that are not in the graph are (quietly) ignored.

`degree_iter(self, nbunch=None, with_labels=False)`

Return iterator that return `degree(n)` or `(n,degree(n))` for all `n` in `nbunch`. If `nbunch` is ommitted, then iterate over all nodes.

Can be called in three ways:

`G.degree_iter(n):` return iterator the degree of node `n`
`G.degree_iter(nbunch):` return a list of values,
one for each `n` in `nbunch` (`nbunch` is any iterable container of nodes.)
`G.degree_iter():` same as `nbunch = all nodes in graph`.

If `with_labels==True`, iterator will return an `(n,degree(n))` tuple of node and degree.

Those nodes in `nbunch` that are not in the graph will be (quietly) ignored.

`delete_edge(self, u, v=None)`

Delete the single edge `(u,v)`.

Can be used in two basic forms: Both `G.delete_edge(u,v)` and `G.delete_edge((u,v))` are equivalent forms of deleting a single edge between nodes `u` and `v`. Return quietly witoutt complaining if the nodes or the edge do not exist.

`delete_edges_from(self, ebunch)`

Delete the edges in `ebunch` from the graph.

`ebunch`: an iterator or iterable of 2-tuples `(u,v)`.

Edges that are not in the graph are ignored.

`delete_node(self, n)`

Delete node `n` from graph.

Attempting to delete a non-existent node will raise an exception.

delete_nodes_from(*self*, *nbunch*)

Remove nodes in *nbunch* from graph.

nbunch:

an iterable or iterator containing valid (hashable) node names.

Attempting to delete a non-existent node will raise an exception.
This could mean some nodes got deleted and other valid nodes did not.

edge_boundary(*self*, *nbunch1*, *nbunch2=None*)

Return list of edges (*n1*,*n2*) with *n1* in *nbunch1* and *n2* in *nbunch2*. If *nbunch2* is omitted or *nbunch2=None*, then *nbunch2* is all nodes not in *nbunch1*.

Nodes in *nbunch1* and *nbunch2* that are not in the graph are ignored.

nbunch1 and *nbunch2* must be disjoint, else raise an exception.

edges(*self*, *nbunch=None*, *with_labels=False*)

Return list of all edges that are adjacent to a node in *nbunch*, or a list of all edges in graph if no nodes are specified.

See `add_node` for definition of *nbunch*.

Those nodes in *nbunch* that are not in the graph will be (quietly) ignored.

with_labels=True option is not supported because in that case you should probably use `neighbors()`.

edges_iter(*self*, *nbunch=None*, *with_labels=False*)

Return iterator that iterates once over each edge adjacent to nodes in *nbunch*, or over all edges in graph if no nodes are specified.

See `add_node` for definition of *nbunch*.

Those nodes in *nbunch* that are not in the graph will be (quietly) ignored.

with_labels=True option is not supported because in that case you should probably use `neighbors()`.

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

Return True if graph contains edge u-v.

has_neighbor(*self*, *u*, *v*=None)

Return True if node u has neighbor v.

has_node(*self*, *n*)

Return True if graph has node n.

(duplicates `self.__contains__`)

"n in G" is a more readable version of "G.has_node(n)"?

is_directed(*self*)

Return True if graph is directed.

neighbors(*self*, *n*, *with_labels*=False)

Return a list of nodes connected to node n.

If *with_labels*=True, return a dict keyed by neighbors.

neighbors.iter(*self*, *n*, *with_labels*=False)

Return an iterator over all neighbors of node n.

If *with_labels*=True, return an iterator of (neighbor, 1) tuples.

node_boundary(*self*, *nbunch1*, *nbunch2*=None)

Return list of all nodes on external boundary of *nbunch1* that are in *nbunch2*. If *nbunch2* is omitted or *nbunch2*=None, then *nbunch2* is all nodes not in *nbunch1*.

Note that by definition the `node_boundary` is external to *nbunch1*.

Nodes in *nbunch1* and *nbunch2* that are not in the graph are ignored.

nbunch1 and *nbunch2* must be disjoint (when restricted to the graph), else a `NetworkXError` is raised.

nodes(*self*)

Return a copy of the graph nodes in a list.

nodes_iter(*self*)

Return an iterator over the graph nodes.

number_of_edges(*self*)

Return the size of a graph = number of edges.

number_of_nodes(*self*)

Return number of nodes.

order(*self*)

Return the order of a graph = number of nodes.

print_dna(*self*)

Print graph "DNA": a dictionary of graph names and properties.

In this version the dna is provided as a user-defined variable and should not be relied on.

size(*self*)

Return the size of a graph = number of edges.

subgraph(*self*, nbunch, inplace=False, create_using=None)

Return the subgraph induced on nodes in nbunch.

nbunch: either a singleton node, a string (which is treated as a singleton node), or any iterable (non-string) container of nodes for which len(nbunch) is defined. For example, a list, dict, set, Graph, numeric array, or user-defined iterable object.

Setting inplace=True will return the induced subgraph in original graph by deleting nodes not in nbunch. This overrides create_using.
Warning: this can destroy the graph.

Unless otherwise specified, return a new graph of the same type as self. Use (optional) create_using=R to return the resulting subgraph in R. R can be an existing graph-like object (to be emptied) or R can be a call to a graph object, e.g. create_using=DiGraph(). See documentation for empty_graph()

Note: use subgraph(G) rather than G.subgraph() to access the more general subgraph() function from the operators module.

`to_directed(self)`

Return a directed representation of the graph G.

A new digraph is returned with the same name, same nodes and with each edge `u-v` represented by two directed edges `u->v` and `v->u`.

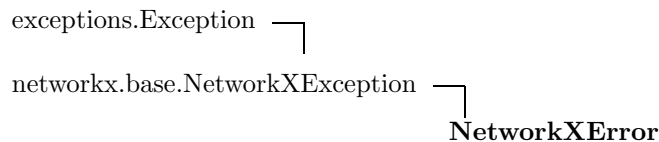
`to_undirected(self)`

Return the undirected representation of the graph G.

This graph is undirected, so merely return a copy.

Inherited from object: `__delattr__`, `__getattr__`, `__hash__`, `__new__`, `__reduce__`, `__reduce_ex__`, `__repr__`, `__setattr__`

2.5 Class `NetworkXError`

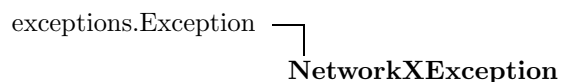


Exception for a serious error in NetworkX

2.5.1 Methods

Inherited from Exception: `__init__`, `__getitem__`, `__str__`

2.6 Class `NetworkXException`



Known Subclasses: `NetworkXError`

Base class for exceptions in NetworkX.

2.6.1 Methods

Inherited from Exception: `__init__`, `__getitem__`, `__str__`

3 Module `networkx centrality`

Centrality measures.

3.1 Functions

`betweenness centrality(G, v=False, cutoff=False, normalized=True)`

Betweenness centrality for nodes.

The fraction of number of shortest paths that go through each node.

```
>>> b=betweenness centrality(G)
```

Returns a dictionary of betweenness values keyed by node.

The betweenness is normalized to be between [0,1].

The algorithm is described in brandes-2003-faster.

If `normalized=False` the resulting betweenness is not normalized.

Reference:

brandes-2003-faster

Ulrik Brandes,

Faster Evaluation of Shortest-Path Based Centrality Indices, 2003,
available at <http://citeseer.nj.nec.com/brandes00faster.html>

`closeness centrality(G, v=False)`

Closeness centrality for nodes (1/average distance to all nodes).

Returns a dictionary of closeness centrality values keyed by node.

The closeness centrality is normalized to be between [0,1].

`degree centrality(G, v=False)`

Degree centrality for nodes (fraction of nodes connected to).

Returns a dictionary of degree centrality values keyed by node.

The degree centrality is normalized to be between [0,1].

```
edge_betweenness(G, nodes=False, cutoff=False)
```

Edge Betweenness

WARNING:

This module is for demonstration and testing purposes.

3.2 Variables

Name	Description
<code>--author--</code>	Value: 'Aric Hagberg (hagberg@lanl.gov)\nPieter Swart - (swart@lanl.gov)' (<i>type=str</i>)
<code>--credits--</code>	Value: '' (<i>type=str</i>)
<code>--date--</code>	Value: '\$Date: 2005-07-06 08:02:28 -0600 (Wed, 06 Jul - 2005) \$' (<i>type=str</i>)
<code>--revision--</code>	Value: '\$Revision: 1064 \$' (<i>type=str</i>)

4 Module `networkx.cliques`

Cliques - Find and manipulate cliques of graphs

Note that finding the largest clique of a graph has been shown to be an NP complete problem so the algorithms here could take a LONG time to run. In practice it hasn't been too bad for the graphs tested.

4.1 Functions

<code>cliques_containing_node(<i>G</i>, <i>nodes</i>=None, <i>cliques</i>=None, <i>with_labels</i>=False)</code>
Returns a list of cliques containing the given node.
Returns a single list or list of lists depending on input nodes.
Returns a dict keyed by node if "with_labels=True".
Optional list of cliques can be input if already computed.

find_cliques(*G*)

Find-cliques algorithm based on Bron & Kerbosch

This algorithm searches for maximal cliques in a graph.
maximal cliques are the largest complete subgraph containing
a given point. The largest maximal clique is sometimes called
the maximum clique.

This algorithm produces the list of maximal cliques each
of which are a list of the members of the clique.

Based on Algol algorithm published by Bron & Kerbosch
A C version is available as part of the rambin package.
<http://www.ram.org/computing/rambin/rambin.html>

Reference:

```
@article{362367,
  author = {Coen Bron and Joep Kerbosch},
  title = {Algorithm 457: finding all cliques of an undirected graph},
  journal = {Commun. ACM},
  volume = {16},
  number = {9},
  year = {1973},
  issn = {0001-0782},
  pages = {575--577},
  doi = {http://doi.acm.org/10.1145/362342.362367},
  publisher = {ACM Press},
}
```

graph_clique_number(*G*, *cliques*=None)

Return the clique number (size the largest clique) for *G*.
Optional list of cliques can be input if already computed.

graph_number_of_cliques(*G*, *cliques*=None)

Returns the number of maximal cliques in *G*
Optional list of cliques can be input if already computed.

make_clique_bipartite(*G*, *fpos*=None, *create_using*=None, ***kws*)

Create a bipartite clique graph from a graph *G*.
 Nodes of *G* are retained as the "bottom nodes" of *B* and
 cliques of *G* become "top nodes" of *B*.
 Edges are present if a bottom node belongs to the clique
 represented by the top node.

Returns a Graph with additional attribute *B.node_type*
 which is "Bottom" or "Top" appropriately.

if *fpos* is not None, a second additional attribute *B.pos*
 is created to hold the position tuple of each node for viewing
 the bipartite graph.

make_max_clique_graph(*G*, *create_using*=None, ***kws*)

Create the maximal clique graph of a graph.
 It 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.

Note: This should be the same as `make_clique_bipartite` followed
 by `project_up`, but it saves all the intermediate stuff.

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

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

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

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

Returns the number of maximal cliques for each node.

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

project_down(*B*, *create_using*=None, ***kws*)

Project a bipartite graph *B* down onto its "Bottom Nodes".
The nodes retain their names and are connected if they
share a common Top Node in the Bipartite Graph.
Returns a Graph.

project_up(*B*, *create_using*=None, ***kws*)

Project a bipartite graph *B* up onto its "Top Nodes".
The nodes retain their names and are connected if they
share a common Bottom Node in the Bipartite Graph.
Returns a Graph.

4.2 Variables

Name	Description
<code>--author--</code>	Value: 'Dan Schult (dschult@colgate.edu)' (<i>type=</i> <code>str</code>)
<code>--credits--</code>	Value: '' (<i>type=</i> <code>str</code>)
<code>--date--</code>	Value: '\$Date: 2005-06-15 07:56:03 -0600 (Wed, 15 Jun - 2005) \$' (<i>type=</i> <code>str</code>)
<code>--revision--</code>	Value: '\$Revision: 1021 \$' (<i>type=</i> <code>str</code>)

5 Module `networkx.cluster`

Compute clustering coefficients and transitivity of graphs.

Clustering coefficient

For each node find the fraction of possible triangles that are triangles,
 $c_i = \text{triangles}_i / (k_i(k_i-1)/2)$
where k_i is the degree of node i .

A coefficient for the whole graph is the average $C = \text{avg}(c_i)$

Transitivity

Find the fraction of all possible triangles which are in fact triangles.
Possible triangles are identified by the number of "triads" (two edges
with a shared vertex)

$T = 3 * \text{triangles} / \text{triads}$

5.1 Functions

<code>average_clustering(<i>G</i>)</code>
Average clustering coefficient for a graph.
Note: this is a space saving routine; It might be faster to use <code>clustering</code> to get a list and then take average.

<code>clustering(<i>G</i>, <i>nbunch</i>=None, **<i>kws</i>)</code>
Clustering coefficient for each node in <i>nbunch</i>

<code>transitivity(<i>G</i>)</code>
Transitivity (fraction of transitive triangles) for a graph

<code>triangles(<i>G</i>, <i>nbunch</i>=None, **<i>kws</i>)</code>
Return number of triangles for <i>nbunch</i> of nodes.
If <i>nbunch</i> is None, then return triangles for every node.
Note: Each triangle is counted three times: once at each vertex.

5.2 Variables

Name	Description
<code>__author__</code>	Value: 'Aric Hagberg (hagberg@lanl.gov)\nPieter Swart - (swart@lanl.gov)\nDan Schult (... (<i>type=str</i>)
<code>__credits__</code>	Value: '' (<i>type=str</i>)
<code>__date__</code>	Value: '\$Date: 2005-06-14 12:48:10 -0600 (Tue, 14 Jun - 2005) \$\n' (<i>type=str</i>)
<code>__revision__</code>	Value: '\$Revision: 1012 \$\n' (<i>type=str</i>)

6 Module networkx.cores

Find and manipulate the k-cores of a graph

6.1 Functions

```
find_cores(G, with_labels=True)
```

Return the core number for each vertex.

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

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

6.2 Variables

Name	Description
<code>__author__</code>	Value: 'Dan Schult(dschult@colgate.edu)' (<i>type=str</i>)
<code>__credits__</code>	Value: '' (<i>type=str</i>)
<code>__date__</code>	Value: '\$Date: 2005-03-30 16:56:28 -0700 (Wed, 30 Mar - 2005) \$' (<i>type=str</i>)
<code>__revision__</code>	Value: '\$Revision: 911 \$' (<i>type=str</i>)

7 Package `networkx.drawing`

7.1 Modules

- **layout:**
Layout (positioning) algorithms for graph drawing.
(Section 8, p. 40)
- **nx_pydot:**
Import and export `networkx` networks to dot format using `pydot`.
(Section 9, p. 42)
- **nx_pylab:**
Draw networks with `matplotlib/pylab`.
(Section 10, p. 44)
- **nx_vtk:**
Draw networks in 3d with `vtk`.
(Section 11, p. 47)

8 Module `networkx.drawing.layout`

Layout (positioning) algorithms for graph drawing.

8.1 Functions

`circular_layout(G, dim=2)`

Circular layout.

Crude version that doesn't try to minimize edge crossings.

`graph_low_ev_pi(uhat, G, eps=0.001, iterations=10000)`

Power Iteration method to find smallest eigenvectors of `Laplacian(G)`.

Note: constant eigenvector has `eigenvalue=0` but is not included in the count of smallest eigenvalues.

`uhat` -- list of `p` initial guesses (dicts) for the `p` eigenvectors.

`G` -- The Graph from which Laplacian is calculated.

`eps` -- tolerance for norm of change in eigenvalue estimate.

`iterations` -- maximum number of iterations to use.

`random_layout(G, dim=2)`

Random layout.

`shell_layout(G, nlist=None, dim=2)`

Shell layout.

Crude version that doesn't try to minimize edge crossings.

`nlist` is an optional list of lists of nodes to be drawn at each shell level. Only one shell with all nodes will be drawn if not specified.

`spectral_layout(G, dim=2, vpos=None, iterations=1000, eps=0.001)`

Return the position vectors for drawing `G` using spectral layout.

`spring_layout(G, iterations=50, dim=2, node_pos=False)`

Spring force model layout

8.2 Variables

Name	Description
<code>--author--</code>	Value: 'Aric Hagberg (hagberg@lanl.gov)\nDan Schult(ds-chult@colgate.edu)' (<i>type=str</i>)
<code>--credits--</code>	Value: '' (<i>type=str</i>)
<code>--date--</code>	Value: '\$Date: 2005-06-15 08:53:26 -0600 (Wed, 15 Jun - 2005) \$' (<i>type=str</i>)
<code>--revision--</code>	Value: '\$Revision: 1033 \$' (<i>type=str</i>)

9 Module `networkx.drawing.nx_pydot`

Import and export `networkx` networks to dot format using `pydot`.

References:

`pydot` Homepage: <http://www.dkbza.org/pydot.html>

Graphviz: <http://www.research.att.com/sw/tools/graphviz/>

DOT Language: <http://www.research.att.com/~erg/graphviz/info/lang.html>

9.1 Functions

<code>networkx_from_pydot(D, result=False)</code>

Creates an <code>networkx</code> graph from an <code>pydot</code> graph <code>D</code>

<code>pydot_from_networkx(N)</code>

Creates a <code>pydot</code> graph from an <code>networkx</code> graph <code>N</code>

<code>pydot_layout(G, **kws)</code>

Create layout using <code>pydot</code> and <code>graphviz</code> . Returns a dictionary of positions keyed by node.

<pre>>>> pos=pydot_layout(G) >>> pos=pydot_layout(G,prog="twopi")</pre>

<code>read_dot(path=False)</code>

Creates an <code>networkx</code> graph from a dot file

<code>write_dot(G, path=False)</code>

Write <code>G</code> to a <code>graphviz</code> dot file.

9.2 Variables

Name	Description
<code>__author__</code>	Value: 'Aric Hagberg (hagberg@lanl.gov)' (<i>type=</i> <code>str</code>)
<code>__credits__</code>	Value: '' (<i>type=</i> <code>str</code>)
<code>__date__</code>	Value: '\$Date: 2005-06-15 08:55:33 -0600 (Wed, 15 Jun - 2005) \$' (<i>type=</i> <code>str</code>)

continued on next page

Name	Description
<code>__revision__</code>	Value: <code>'\$Revision: 1034 \$'</code> (<i>type=</i> <code>str</code>)

10 Module `networkx.drawing.nx_pylab`

Draw networks with matplotlib/pylab.

References:

matplotlib: <http://matplotlib.sourceforge.net/>

10.1 Functions

`draw(G, **kwargs)`

Draw networkx graph using spring layout.

NB:

`pylab.draw()` has been renamed to `pylab.redraw()`
so use `redraw()` from pylab interface.

`draw_circular(G, **kwargs)`

Draw networkx graph in circular layout

`draw_nx(G, node_pos, **kwargs)`

Draw networkx graph with nodes at `node_pos`.

See `layout.py` for functions that compute node positions.

`node_pos` is a dictionary keyed by vertex with a two-tuple
of x-y positions as the value.

Use kwarg of `node_color` with a dictionary keyed by vertex with a
floating point number as a value.

Use kwarg of `node_size` with a dictionary keyed by vertex with a
floating point number as a value.

`draw_nxpydot(G, **kws)`

Draw networkx graph using pydot and graphviz layout.

```
>>> G=barbell_graph(5,10)
>>> d=G.degree(with_labels=True)
>>> draw_nxpydot(G)
>>> draw_nxpydot(G,node_color=d,cmap=cm.pink)
>>> draw_nxpydot(G,prog='neato')
```

Can use `prog= "neato", "dot", "circo", "twopi", or "fdp"`

draw_nxpydot_nolabels(*G*, ****kwargs**)

Draw networkx graph without labels in textbook style.

`node_color` is black, unless specified in `kwargs`
(with `node_color=` a dictionary that maps each key=node to a floating point number.)

e.g.

```
>>> G=star_graph(10)
>>> d=G.degree(with_labels=True)
>>> draw_nxpydot_nolabels(G)
>>> draw_nxpydot_nolabels(G,node_color=d,cmap=cm.pink)
>>> draw_nxpydot_nolabels(G,prog='neato')
```

Can use `prog=` "neato", "dot", "circo", "twopi", or "fdp"

draw_pydot(*P*, ****kwargs**)

Draw pydot graph *P* with matplotlib.

The pydot graph must have position information in graphviz dot format.

draw_pydot_subgraph(*ax*, *P*, ****kws**)

Draw a pydot network *P* that has nested subgraphs.

draw_random(*G*, ****kwargs**)

Draw networkx graph with random layout.

draw_shell(*G*, ****kwargs**)

Draw networkx graph with shell layout

draw_spectral(*G*, ****kwargs**)

Draw networkx graph with spectral layout.

draw_spring(*G*, ****kwargs**)

Draw networkx graph with spring layout

drawg(*G*, ****kwargs**)

Draw networkx graph using spring layout

```
mpl_network(ax, node_pos, edge_pos, node_size=300, node_color='r', node_marker='o',
node_labels=True, fonts={}, edge_color='k', edge_width=1.0, cmap=None, norm=None, vmin=None,
vmax=None, alpha=1.0, digraph=False)
```

Draw network with matplotlib using node positions=`node_pos`
and edge positions=`edge_pos`.

Steals heavily from `matplotlib.axes.scatter`.

10.2 Variables

Name	Description
<code>--author--</code>	Value: 'Aric Hagberg (hagberg@lanl.gov)' (<i>type=str</i>)
<code>--credits--</code>	Value: '' (<i>type=str</i>)
<code>--date--</code>	Value: '\$Date: 2005-06-15 11:29:39 -0600 (Wed, 15 Jun - 2005) \$' (<i>type=str</i>)
<code>--revision--</code>	Value: '\$Revision: 1035 \$' (<i>type=str</i>)

11 Module `networkx.drawing.nx_vtk`

Draw networks in 3d with `vtk`.

References:

`vtk`: <http://www.vtk.org/>

11.1 Functions

`draw_nxvtk(G, node_pos)`

Draw `networkx` graph in 3d with nodes at `node_pos`.

See `layout.py` for functions that compute node positions.

`node_pos` is a dictionary keyed by vertex with a three-tuple of x-y positions as the value.

The node color is plum.

The edge color is banana.

All the nodes are the same size.

11.2 Variables

Name	Description
<code>__author__</code>	Value: 'Aric Hagberg (hagberg@lanl.gov)' (<i>type=str</i>)
<code>__credits__</code>	Value: '' (<i>type=str</i>)
<code>__date__</code>	Value: '\$Date: 2005-06-17 08:10:29 -0600 (Fri, 17 Jun - 2005) \$' (<i>type=str</i>)
<code>__revision__</code>	Value: '\$Revision: 1051 \$' (<i>type=str</i>)

12 Package `networkx.generators`

A package for generating various graphs in `networkx`.

12.1 Modules

- **atlas:**
Generators for the small graph atlas.
(Section 13, p. 49)
- **classic:**
Generators for some classic graphs.
(Section 14, p. 50)
- **degree_seq:**
Generate graphs with a given degree sequence.
(Section 15, p. 55)
- **geometric:**
Generators for geometric graphs.
(Section 16, p. 59)
- **random_graphs:**
Generators for random graphs
(Section 17, p. 60)
- **small:**
Various small and named graphs, together with some compact generators.
(Section 18, p. 66)

13 Module `networkx.generators.atlas`

Generators for the small graph atlas.

See

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

13.1 Functions

`graph_atlas_g()`

Return the list `[G1,G2,...,G1252]` of graphs as named in the Graph Atlas.
`G1,...,G1252` are all graphs with up to 7 nodes.

The graphs are listed:

in increasing order of number of nodes;

for a fixed number of nodes,
in increasing order of the number of edges;

for fixed numbers of nodes and edges,
in increasing order of the degree sequence,
for example `111223 < 112222`;

for fixed degree sequence, in increasing number of automorphisms.

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

13.2 Variables

Name	Description
<code>__author__</code>	Value: <code>'Pieter Swart (swart@lanl.gov)'</code> (<i>type=</i> <code>str</code>)
<code>__credits__</code>	Value: <code>''</code> (<i>type=</i> <code>str</code>)
<code>__date__</code>	Value: <code>'\$Date: 2005-03-30 16:56:28 -0700 (Wed, 30 Mar - 2005) \$'</code> (<i>type=</i> <code>str</code>)
<code>__revision__</code>	Value: <code>'\$Revision: 911 \$'</code> (<i>type=</i> <code>str</code>)

14 Module `networkx.generators.classic`

Generators for some classic graphs.

The typical graph generator is called as follows:

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

returning the complete graph on n nodes labeled $1, \dots, 100$ as a simple graph. Except for `empty_graph`, all these generators return a Graph class (i.e. a simple undirected graph).

14.1 Functions

<code>balanced_tree(r, h)</code>
Return the perfectly balanced r -tree of height h .
For $r \geq 2$, $h \geq 1$, this is the rooted tree where all leaves are at distance h from the root. The root has degree r and all other internal nodes have degree $r+1$. Graph order= $1+r+r^2+\dots+r^h=(r^{h+1}-1)/(r-1)$, graph size=order-1. Node labels are integers numbered 1 (the root) up to order.

<code>barbell_graph($m1, m2$)</code>
Return the Barbell Graph: two complete graphs connected by a path.
For $m1 > 1$ and $m2 \geq 0$.
Two complete graphs K_{m1} form the left and right bells, and are connected by a path P_{m2} . The $2*m1+m2$ nodes are numbered $1, \dots, m1$ for the left barbell, $m1+1, \dots, m1+m2$ for the path, and $m1+m2+1, \dots, 2*m1+m2$ for the right barbell. The 3 subgraphs are joined via the edges $(m1, m1+1)$ and $(m1+m2, m1+m2+1)$. If $m2=0$, this is merely two complete graphs joined together. This graph is an extremal example in David Aldous and Jim Fill's etext on Random Walks on Graphs.

`circular_ladder_graph(n)`

Return the circular ladder graph CL_n of length n .

CL_n consists of two concentric n -cycles in which each of the n pairs of concentric nodes are joined by an edge.

`complete_bipartite_graph($n1, n2$)`

Return the complete bipartite graph $K_{\{n1, n2\}}$.

Contains $n1$ nodes in the first subgraph and $n2$ nodes in the second subgraph.

`complete_graph(n)`

Return the Complete graph K_n with n nodes.

`cycle_graph(n)`

Return the cycle graph C_n over n nodes.

C_n is P_n with two end-nodes connected.

`dorogovtsev_goltsev_mendes_graph(n)`

Return the hierarchically constructed Dorogovtsev-Goltsev-Mendes graph.

n is the generation.

See: [arXiv:/cond-mat/0112143](https://arxiv.org/abs/cond-mat/0112143) by Dorogovtsev, Goltsev and Mendes.

`empty_graph(n=0, create_using=None, **kws)`

Return the empty graph with n nodes
(with integer labels $1, \dots, n$) and zero edges.

```
>>> G=empty_graph(n)
```

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.).

Firstly, the variable `create_using` can be used to create an empty digraph, network, etc. For example,

```
>>> G=empty_graph(n,create_using=DiGraph())
```

will create an empty digraph on n nodes, and

```
>>> G=empty_graph(n,create_using=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 `baseNX`) and then add n nodes and zero edges, and return the modified graph (resp. digraph, pseudograph, etc.).

WARNING: The graph `dna` is not scrubbed in this process.

See also `create_empty_copy(G)`.

`grid_2d_graph(m, n)`

Return the 2d grid graph of $m \times n$ nodes,
each connected to its nearest neighbors.

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.

hypercube_graph(*n*)

return the n-dimensional hypercube.

ladder_graph(*n*)

Return the Ladder graph of length *n*.

This is two rows of *n* nodes,
each pair connected by a single edge.

lollipop_graph(*m*, *n*)

Return the Lollipop Graph; K_m connected to P_n .

This is the Barbell Graph without the right barbell.

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

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

null_graph(*create_using*=None, ***kws*)

Return the Null graph with no nodes or edges.

path_graph(*n*)

Return the Path graph P_n of *n* nodes linearly connected by *n*-1 edges.

`periodic_grid_2d_graph(m, n)`

Return the 2-D Grid Graph of $m \times n$ nodes,
each connected to its nearest neighbors.

Boundary nodes are identified in a periodic fashion.

`star_graph(n)`

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

`trivial_graph()`

Return the Trivial graph with one node (with integer label 1)
and no edges.

`wheel_graph(n)`

Return the wheel graph: a single hub node connected
to each node of the $(n-1)$ -node cycle graph.

14.2 Variables

Name	Description
<code>__author__</code>	Value: 'Aric Hagberg (hagberg@lanl.gov)\nPieter Swart - (swart@lanl.gov)' (<i>type=str</i>)
<code>__credits__</code>	Value: '' (<i>type=str</i>)
<code>__date__</code>	Value: '\$Date: 2005-06-17 14:06:03 -0600 (Fri, 17 Jun - 2005) \$' (<i>type=str</i>)
<code>__revision__</code>	Value: '\$Revision: 1056 \$' (<i>type=str</i>)

15 Module `networkx.generators.degree_seq`

Generate graphs with a given degree sequence.

15.1 Functions

`configuration_model(deg_sequence, seed=None)`

Return a pseudograph with given degree sequence.

`deg_sequence`: degree sequence, a list of integers with each entry

corresponding to the degree of a node (need not be sorted). A non-graphical degree sequence (i.e. one not realizable by some simple graph) will raise an `Exception`.

`seed`: seed for random number generator (default=None)

Steps:

Check if `deg_sequence` is a valid degree sequence.

Create `N` nodes with stubs of given degree.

Randomly select two available stubs and connect them with an edge.

As described by Newman [newman-2003-structure].

Nodes are labeled `1, ..., len(deg_sequence)`, corresponding to their position in `deg_sequence`.

This process can lead to duplicate edges and loops, and therefore returns a pseudograph type. You can call `remove_parallel()` and `remove_selfloops()` to get a simple graph (but likely without the exact specified degree sequence). This "finite-size effect" decreases as the size of the graph increases.

References:

[newman-2003-structure] M.E.J. Newman, "The structure and function of complex networks", SIAM REVIEW 45-2, pp 167-256, 2003.

`create_degree_sequence(n, sfunction=None, max_tries=50, **kws)`

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

`n`: length of degree sequence = number of nodes

`sfunction`: a function, called as "`sfunction(n,kws)`",

that returns a list of `n` real or integer values.

`max_tries`: max number of attempts at creating valid degree sequence.

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

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

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

`degree_sequence_tree(deg_sequence)`

Make a tree for the given degree sequence.

A tree has `#nodes-#edges=1` so the degree sequence must have `len(deg_sequence)-sum(deg_sequence)/2=1`

havel_hakimi_graph(*deg_sequence*, *seed*=None)

Return a simple graph with given degree sequence, constructed using the Havel-Hakimi algorithm.

deg_sequence: degree sequence, a list of integers with each entry

corresponding to the degree of a node (need not be sorted).

A non-graphical degree sequence (not sorted).

A non-graphical degree sequence (i.e. one not realizable by some simple graph) raises an Exception.

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

The Havel-Hakimi algorithm constructs a simple graph by successively connecting the node of highest degree to other nodes of highest degree, resorting remaining nodes by degree, and repeating the process. The resulting graph has a high degree-associativity. Nodes are labeled 1,..., len(*deg_sequence*), corresponding to their position in *deg_sequence*.

See Theorem 1.4 in [chartrand-graphs-1996].

This algorithm is also used in the function `is_valid_degree_sequence`.

References:

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

`is_valid_degree_sequence(deg_sequence)`

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

`deg_sequence`: degree sequence, a list of integers with each entry

corresponding to the degree of a node (need not be sorted).

A non-graphical degree sequence (i.e. one not realizable by some simple graph) will raise an exception.

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

References:

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

15.2 Variables

Name	Description
<code>--author--</code>	Value: 'Aric Hagberg (hagberg@lanl.gov)\nPieter Swart - (swart@lanl.gov)\nDan Schult (... (<i>type=str</i>)
<code>--credits--</code>	Value: '' (<i>type=str</i>)
<code>--date--</code>	Value: '\$Date: 2005-06-15 12:42:59 -0600 (Wed, 15 Jun - 2005) \$\n' (<i>type=str</i>)
<code>--revision--</code>	Value: '\$Revision: 1037 \$\n' (<i>type=str</i>)

16 Module `networkx.generators.geometric`

Generators for geometric graphs.

16.1 Functions

`random_geometric_graph`(*n*, *radius*, *result=False*, ***kws*)

Random geometric graph in the unit cube

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

16.2 Variables

Name	Description
<code>__author__</code>	Value: 'Aric Hagberg (hagberg@lanl.gov)\nPieter Swart - (swart@lanl.gov)' (<i>type=str</i>)
<code>__credits__</code>	Value: '' (<i>type=str</i>)
<code>__date__</code>	Value: '\$Date: 2005-06-15 12:44:45 -0600 (Wed, 15 Jun - 2005) \$' (<i>type=str</i>)
<code>__revision__</code>	Value: '\$Revision: 1038 \$' (<i>type=str</i>)

17 Module `networkx.generators.random_graphs`

Generators for random graphs

17.1 Functions

`barabasi_albert_graph`(*n*, *m*, *seed*=None)

Return random graph using Barabasi-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.

The initialization is a graph with with *m* nodes.

Reference:

```
@ARTICLE{barabasi-1999-emergence,
  TITLE = {Emergence of scaling in random networks},
  AUTHOR = {A. L. Barabasi and R. Albert},
  JOURNAL = {SCIENCE},
  VOLUME = {286},
  NUMBER = {5439},
  PAGES = {509 -- 512},
  MONTH = {OCT},
  YEAR = {1999},
}
```

Parameters

n: the number of nodes
m: average degree and must be an positive integer
seed: seed for random number generator (default=None)

`binomial_graph`(*n*, *p*, *seed*=None)

Return a binomial random graph $G_{\{n,p\}}$.

Parameters

n: the number of nodes
p: probability that any given edge exist
seed: seed for random number generator (default=None)

erdos_renyi_graph(*n*, *m*, *seed*=None)

Return the Erdos-Renyi random graph $G_{\{n,m\}}$.**Parameters**

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

newman_watts_strogatz_graph(*n*, *k*, *p*, *seed*=None)

Return a Newman-Watts-Strogatz small world graph.

The graph is a ring with *k* neighbors with new edges (shortcuts) added randomly with probability *p* for each edge. No edges are removed.

Parameters

n: the number of nodes
k: each vertex is connected to *k* neighbors in the circular topology
p: the probability of adding a new edge for each edge
seed: seed for random number generator (default=None)

powerlaw_cluster_graph(*n, m, p, seed=None*)

Holme and Kim algorithm for growing graphs with powerlaw degree distribution and approximate average clustering.

Reference:

```
@Article{growing-holme-2002,
author = {P. Holme and B. J. Kim},
title = {Growing scale-free networks with tunable clustering},
journal = {Phys. Rev. E},
year = {2002},
volume = {65},
number = {2},
pages = {026107},
}
```

The average clustering has a hard time getting above a certain cutoff that depends on *m*. This cutoff is often quite low.

It is essentially the Barabasi-Albert growth model with an extra step that each random edge is followed by a chance of making an edge to one of its neighbors too (and thus a triangle).

This algorithm improves on B-A in the sense that it enables a higher average clustering to be attained if desired. The largest average clustering seems to be independent of *n* and attained with *m*=1 and *p*=1 (*cc*=0.74 or so).

Parameters

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

random_lobster(*n, p1, p2, seed=None*)

Return a random lobster.

A caterpillar is a tree that reduces to a path graph when pruning all leave nodes. A lobster is a tree that reduces to a caterpillar when pruning all leave nodes.

Parameters

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

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

Return a tree with a powerlaw degree distribution.

A trial powerlaw degree sequence is chosen and then elements are swapped with new elements from a powerlaw distribution until the sequence makes a tree (`#edges=#nodes-1`).

Parameters

`n`: the number of nodes
`gamma`: exponent of power law is `gamma`
`seed`: seed for random number generator (default=None)
`tries`: number of attempts to adjust sequence to make a tree

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

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

A trial powerlaw degree sequence is chosen and then elements are swapped with new elements from a powerlaw distribution until the sequence makes a tree (`#edges=#nodes-1`).

Parameters

`n`: the number of nodes
`gamma`: exponent of power law is `gamma`
`seed`: seed for random number generator (default=None)
`tries`: number of attempts to adjust sequence to make a tree

`random_regular_graph(d, n, seed=None)`

Return a random regular graph of n nodes each with degree d , $G_{\{n,d\}}$.
 Return False if unsuccessful.

$n*d$ must be even

Nodes are numbered $0 \dots n-1$.

To get a uniform sample from the space of random graphs
 you should chose $d < n^{\{1/3\}}$.

.

For algorithm see Kim and Vu's paper.

Reference:

```
@inproceedings{kim-2003-generating,
author = {Jeong Han Kim and Van H. Vu},
title = {Generating random regular graphs},
booktitle = {Proceedings of the thirty-fifth ACM symposium on Theory of computing},
year = {2003},
isbn = {1-58113-674-9},
pages = {213--222},
location = {San Diego, CA, USA},
doi = {http://doi.acm.org/10.1145/780542.780576},
publisher = {ACM Press},
}
```

The algorithm is based on an earlier paper:

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


`random_shell_graph`(*constructor*, *seed*=None)

Return a random shell graph for the constructor given.

constructor: a list of three-tuples [(n1,m1,d1),(n2,m2,d2),...]
one for each shell, starting at the center shell.

n : the number of nodes in the shell

m : the number or edges in the shell

d

the ratio of inter (next) shell edges to intra shell edges.

d=0 means no intra shell edges.

d=1 for the last shell

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

```
>>> constructor=[(10,20,0.8),(20,40,0.8)]
```

```
>>> G=random_shell_graph(constructor)
```

`watts_strogatz_graph`(*n*, *k*, *p*, *seed*=None)

Return a Watts-Strogatz small world graph.

The graph is a ring with *k* neighbors with
edges rewired randomly with probability *p*.

Parameters

n: the number of nodes

k: each vertex is connected to *k* neighbors in the circular topology

p: the probability of rewiring an edge

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

17.2 Variables

Name	Description
<code>__author__</code>	Value: 'Aric Hagberg (hagberg@lanl.gov)\nPieter Swart - (swart@lanl.gov)\nDan Schult(d... (<i>type=</i> str)
<code>__credits__</code>	Value: '' (<i>type=</i> str)
<code>__date__</code>	Value: '\$Date: 2005-06-17 08:06:22 -0600 (Fri, 17 Jun - 2005) \$' (<i>type=</i> str)
<code>__revision__</code>	Value: '\$Revision: 1049 \$' (<i>type=</i> str)

18 Module `networkx.generators.small`

Various small and named graphs, together with some compact generators.

18.1 Functions

<code>bull_graph()</code>
Return the Bull graph.

<code>chvatal_graph()</code>
Return the Chvatal graph.

<code>cubical_graph()</code>
Return the 3-regular Platonic Cubical graph.

<code>desargues_graph()</code>
Return the Desargues graph.

<code>diamond_graph()</code>
Return the Diamond graph.

<code>dodecahedral_graph()</code>
Return the Platonic Dodecahedral graph.

<code>frucht_graph()</code>
Return the Frucht Graph.
The Frucht Graph is the smallest cubical graph whose automorphism group consists only of the identity element.

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

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

house_x_graph()

Return the House graph with a cross inside the house square.

icosahedral_graph()

Return the Platonic Icosahedral graph.

krackhardt_kite_graph()

Return the Krackhardt Kite Social Network.

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.

LCF_graph(*n, shift_list, repeats*)

Return the cubic graph specified in LCF notation.

LCF notation (LCF=Lederberg-Coxeter-Fruchte) is a compressed notation used in the generation of various cubic Hamiltonian graphs of high symmetry. See, for example, `dodecahedral_graph`, `desargues_graph`, `heawood_graph` and `pappus_graph` below.

n (number of nodes)

The starting graph is the *n*-cycle with nodes 1,...,*n*.
(The null graph is returned if *n* < 0.)

shift_list = [*s*₁,*s*₂,...,*s*_{*k*}], 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 *v*₁ cycling through the *n*-cycle a total of *k***repeats* with *shift* cycling through *shiftlist* *repeats* times connect *v*₁ with *v*₁+*shift* mod *n*

The utility graph $K_{\{3,3\}}$

```
>>> G=LCF_graph(6,[3,-3],3)
```

The Heawood graph

```
>>> G=LCF_graph(14,[5,-5],7)
```

See <http://mathworld.wolfram.com/LCFNotation.html> for a description and references.

make_small_graph(*graph_description*, *create_using*=None, ****kws**)

Return the small graph described by *graph_description* and *kws*.

graph_description is a list of the form [type,name,n,xlist]

Here type is one of "adjacencylist" or "edgelist",

name is the name of the graph and n the number of nodes.

This constructs a graph of n nodes with integer labels 1,...,n.

If type="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 C₄ can be obtained by

```
>>> G=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=make_small_graph(["adjacencylist","C_4",4,[[2,4],[3],[4],[ ]]])
```

If type="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 C₄ can be obtained by

```
>>> G=make_small_graph(["edgelist","C_4",4,[[1,2],[3,4],[2,3],[4,1]])
```

Other graph descriptors can be passed to Graph() using *kws*

moebius_kantor_graph()

Return the Moebius-Kantor graph.

octahedral_graph()

Return the Platonic Octahedral graph.

pappus_graph()

Return the Pappus graph.

petersen_graph()

Return the Petersen graph.

`sedgewick_maze_graph()`

Return a small maze with a cycle.

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

`tetrahedral_graph()`

Return the 3-regular Platonic Tetrahedral graph.

`truncated_cube_graph()`

Return the skeleton of the truncated cube.

`truncated_tetrahedron_graph()`

Return the skeleton of the truncated Platonic tetrahedron.

`tutte_graph()`

Return the Tutte graph.

18.2 Variables

Name	Description
<code>__author__</code>	Value: 'Aric Hagberg (hagberg@lanl.gov)\nPieter Swart - (swart@lanl.gov)' (<i>type=str</i>)
<code>__credits__</code>	Value: '' (<i>type=str</i>)
<code>__date__</code>	Value: '\$Date: 2005-06-15 12:53:08 -0600 (Wed, 15 Jun - 2005) \$' (<i>type=str</i>)
<code>__revision__</code>	Value: '\$Revision: 1040 \$' (<i>type=str</i>)

19 Module networkx.hybrid

Hybrid

19.1 Functions

is_kl_connected(*G*, *k*, *l*, ****kws**)

Returns True if *G* is kl connected

kl_connected_subgraph(*G*, *k*, *l*, ****kws**)

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

19.2 Variables

Name	Description
<code>__author__</code>	Value: 'Aric Hagberg (hagberg@lanl.gov)\nDan Schult (dschult@colgate.edu)' (<i>type=</i> str)
<code>__credits__</code>	Value: '' (<i>type=</i> str)
<code>__date__</code>	Value: '\$Date: 2005-03-30 16:56:28 -0700 (Wed, 30 Mar - 2005) \$' (<i>type=</i> str)
<code>__revision__</code>	Value: '\$Revision: 911 \$' (<i>type=</i> str)

20 Module `networkx.io`

Read and write graphs and networks.

The example undirected graph below consists of the two edges (a,b),(a,c).

20.1 Functions

```
read_adjlist(path=False, create_using=None)
```

Read graph in single line adjacency list format from path.
The default is to create a simple graph from the adjacency list.
The optional `create_using` argument allows other types of graphs.

```
>>> G=DiGraph()  
>>> G=read_adjlist(file, create_using=G)
```

Example adjacency list file format:

```
# node degree  
a b c  
b a  
c a
```

```
read_edgelist(path=False, create_using=None)
```

Read graph in edgelist format

Example adjacency list file format:

```
# node degree  
a b  
a c
```

```
read_gpickle(path=False)
```

Read graph object in python pickle format
See `cPickle`.

read_multiline_adjlist(*path=False, create_using=None*)

Read graph in multiline adjacency list format.

Example multiline adjacency list file format:

```
# node degree
a 2
b
c
b 1
a
c 1
a
```

write_adjlist(*G, path=False*)

Write graph in single line adjacency list format in file path.

If no file is given, write to standard output.

Example adjacency list file format:

```
# node degree
a b c
b a
c a
```

write_edgelist(*G, path=False*)

Write graph *G* in edgelist format on file path.

If no file is given write to standard output.

Example adjacency list file format:

```
# node degree
a b
a c
```

write_gpickle(*G, path=False*)

Write graph object in python pickle format
See `cPickle`.

```
write_multiline_adjlist(G, path=False)
```

Write graph in multiline adjacency list format.

Example multiline adjacency list file format:

```
# node degree
a 2
b
c
b 1
a
c 1
a
```

20.2 Variables

Name	Description
<code>__author__</code>	Value: 'Aric Hagberg (hagberg@lanl.gov)\nDan Schult (d-schult@colgate.edu)' (<i>type=str</i>)
<code>__credits__</code>	Value: '' (<i>type=str</i>)
<code>__date__</code>	Value: '\$Date: 2005-07-06 07:58:26 -0600 (Wed, 06 Jul - 2005) \$' (<i>type=str</i>)
<code>__revision__</code>	Value: '\$Revision: 1063 \$' (<i>type=str</i>)

21 Module `networkx.isomorph`

Fast checking to see if graphs are not isomorphic.

This isn't a graph isomorphism checker.

21.1 Functions

`fast_graph_could_be_isomorphic(G1, G2)`

Returns False if graphs *G1* and *G2* are definitely not isomorphic.
True does NOT guarantee isomorphism.

Checks for matching degree and triangle sequences.

`faster_graph_could_be_isomorphic(G1, G2)`

Returns False if graphs *G1* and *G2* are definitely not isomorphic.
True does NOT guarantee isomorphism.

Checks for matching degree sequences in *G1* and *G2*.

`graph_could_be_isomorphic(G1, G2)`

Returns False if graphs *G1* and *G2* are definitely not isomorphic.
True does NOT guarantee isomorphism.

Checks for matching degree, triangle, and number of cliques sequences.

21.2 Variables

Name	Description
<code>__author__</code>	Value: 'Pieter Swart (swart@lanl.gov)\nDan Schult (dsc-hult@colgate.edu)' (<i>type=</i> <i>str</i>)
<code>__credits__</code>	Value: '' (<i>type=</i> <i>str</i>)
<code>__date__</code>	Value: '\$Date: 2005-05-31 17:00:13 -0600 (Tue, 31 May - 2005) \$' (<i>type=</i> <i>str</i>)
<code>__revision__</code>	Value: '\$Revision: 1002 \$' (<i>type=</i> <i>str</i>)

22 Module `networkx.operators`

Operations on graphs; including `union`, `complement`, `subgraph`.

22.1 Functions

`cartesian_product(G, H)`

Return the Cartesian product of `G` and `H`.

Tested only on `Graph` class.

`complement(G, create_using=None, **kws)`

Return graph complement of `G`.

Unless otherwise specified, return a new graph of the same type as `self`. Use (optional) `create_using=R` to return the resulting subgraph in `R`. `R` can be an existing graph-like object (to be emptied) or `R` can be a call to a graph object, e.g. `create_using=DiGraph()`. See documentation for `empty_graph()`

Implemented for `Graph`, `DiGraph`, `XGraph`, `XDiGraph`.

Note that `complement()` is not well-defined for `XGraph` and `XDiGraph` objects that allow multiple edges or self-loops.

`compose(G, H, create_using=None, **kws)`

Return a new graph of `G` composed with `H`.

The node sets of `G` and `H` need not be disjoint.

A new graph is returned, of the same class as `G`.

It is recommended that `G` and `H` be either both directed or both undirected.

Optional `create_using=R` returns graph `R` filled in with the `compose(G,H)`. Otherwise 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.

Implemented for `Graph`, `DiGraph`, `XGraph`, `XDiGraph`

convert_node_labels_to_integers(*G*, *first_label*=1, *ordering*='default', *discard_old_labels*=True)

Return a copy of *G*, with *n* node labels replaced with integers, starting at *first_label*.

first_label: (optional, default=1)

An integer specifying the offset in numbering nodes.

The *n* new integer labels are numbered *first_label*, ..., *n*+*first_label*.

ordering: (optional, default="default")

specifies how nodes are ordered. Possible values: "default" (inherit from *G*), "increasing degree", or "decreasing degree"

discard_old_labels

if True (default) discard old labels

if False, create a dict *self*.*node_labels* that maps new labels to old labels, and set *self*.*dna*["node_labeled"]=True

Works for Graph, DiGraph, XGraph, XDiGraph

convert_to_directed(*G*)

Return a new directed representation of the graph *G*.

Works for Graph, DiGraph, XGraph, XDiGraph.

Note: *convert_to_directed*(*G*)=*G*.*to_directed*()

convert_to_undirected(*G*)

Return a new undirected representation of the graph *G*.

Works for Graph, DiGraph, XGraph, XDiGraph.

Note: *convert_to_undirected*(*G*)=*G*.*to_undirected*()

create_empty_copy(*G*)

Return a new, empty graph-like object of the same type/class as *G*.

Works for Graph, DiGraph, XGraph, XDiGraph

disjoint_union(*G, H*)

Return the disjoint union of graphs *G* and *H*, forcing distinct integer node labels.

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.

Implemented for *Graph*, *DiGraph*, *XGraph*, *XDiGraph*.

subgraph(*G, nbunch, inplace=False, create_using=None*)

Return the subgraph induced on nodes in *nbunch*.

nbunch: either a singleton node, a string (which is treated as a singleton node, or any iterable (non-string) container of nodes for which `len(nbunch)` is defined. For example, a list, dict, set, *Graph*, numeric array, or user-defined iterable object.

Setting `inplace=True` will return induced subgraph in original graph by deleting nodes not in *nbunch*.

Unless otherwise specified, return a new graph of the same type as *self*. Use (optional) `create_using=R` to return the resulting subgraph in *R*. *R* can be an existing graph-like object (to be emptied) or *R* is a call to a graph object, e.g. `create_using=DiGraph()`. See documentation for `empty_graph`.

Implemented for *Graph*, *DiGraph*, *XGraph*, *XDiGraph*

Note: `subgraph(G)` calls `G.subgraph()`

`union(G, H, create_using=None, rename=False, **kws)`

Return the union of graphs G and H.

Graphs G and H must be disjoint, otherwise an exception is raised.

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".

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

Optional `create_using=R` returns graph R filled in with the union of G and H. Otherwise 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.

A new name can be specified in the form
`X=graph.union(G,H,name="new_name")`

Implemented for Graph, DiGraph, XGraph, XDiGraph.

22.2 Variables

Name	Description
<code>__author__</code>	Value: 'Aric Hagberg (hagberg@lanl.gov)\nPieter Swart - (swart@lanl.gov)\nDan Schult(d... <i>(type=str)</i>
<code>__credits__</code>	Value: '' <i>(type=str)</i>
<code>__date__</code>	Value: '\$Date: 2005-06-15 08:09:52 -0600 (Wed, 15 Jun - 2005) \$' <i>(type=str)</i>
<code>__revision__</code>	Value: '\$Revision: 1024 \$' <i>(type=str)</i>

23 Module `networkx.paths`

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

23.1 Functions

`center(G, e=None)`

Center of graph.

Nodes with eccentricity equal to radius.

`diameter(G, e=None)`

Diameter of graph.

Maximum of all pairs shortest path.

`dijkstra(G, source, target=None)`

Dijkstra's algorithm for shortest paths in a weighted graph.

See

`dijkstra_path()` - shortest path list of nodes
`dijkstra_path_length()` - shortest path length

Returns a tuple of two dictionaries keyed by node.

The first stores distance from the source.

The second stores the path from the source to that node.

Distances are calculated as sums of weighted edges traversed.

Edges must hold numerical values for `XGraph` and `XDiGraphs`.

The weights are 1 for `Graphs` and `DiGraphs`.

Optional target argument stops the search when target is found.

Based on 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).

dijkstra_path(*G, source, target=None*)

Returns the shortest path for a weighted graph using Dijkstra's algorithm.

The path is computed from the source to an optional target. If a target is specified the path is returned as a list of nodes. If the target is not specified a dictionary of path lists keyed by target node is returned.

Edge data must be numerical values for XGraph and XDiGraphs. The weights are assigned to be 1 for Graphs and DiGraphs.

See also "dijkstra" for more information about the algorithm.

dijkstra_path_length(*G, source, target=None*)

Returns the shortest path length for a weighted graph using Dijkstra's algorithm .

The path length is computed from the source to an optional target. If a target is specified the length is returned as an integer. If the target is not specified a dictionary of path lengths keyed by target node is returned.

Edge data must be numerical values for XGraph and XDiGraphs. The weights are assigned to be 1 for Graphs and DiGraphs.

See also "dijkstra" for more information about the algorithm.

eccentricity(*G, v=None, sp=None, **kws*)

Eccentricity of node v.
Maximum of shortest paths to all other nodes.

If kws with_labels=True
return dict of eccentricities keyed by vertex.

is_directed_acyclic_graph(*G*)

Test if a graph is a directed acyclic graph (DAG).

Return True if G is a DAG. False if not.

periphery(*G, e=None*)

Periphery of graph.
Nodes with eccentricity equal to diameter.

radius(*G*, *e=None*)

Radius of graph

Minimum of all pairs shortest path.

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

Returns list of nodes in a shortest path between source and target (there might be more than one).

If no target is specified, returns dict of lists of paths from source to all nodes.

Cutoff is a limit on the number of hops traversed.

shortest_path_length(*G*, *source*, *target=None*)

Shortest path length from source to target.

topological_sort(*G*)

Return a list of nodes of the graph *G* in topological sort order.A topological sort is a nonunique permutation of the nodes such that an edge from *u* to *v* implies that *u* appears before *v* in the topological sort order.If *G* is not a directed acyclic graph no topological sort exists and the Python keyword `None` is returned.This algorithm is based on a description and proof at <http://www2.toki.or.id/book/AlgDesignManual/book/book2/node70.htm>See also `is_directed_acyclic_graph()`

topological_sort_recursive(*G*)

Return a list of nodes of the graph *G* in topological sort order.

This is a recursive version of topological sort.

23.2 Variables

Name	Description
<code>--author--</code>	Value: 'Aric Hagberg (hagberg@lanl.gov)\nDan Schult(ds-chult@colgate.edu)' (<i>type=</i> <code>str</code>)
<code>--credits--</code>	Value: '' (<i>type=</i> <code>str</code>)

continued on next page

Name	Description
<code>__date__</code>	Value: '\$Date: 2005-06-16 14:29:18 -0600 (Thu, 16 Jun - 2005) \$' (<i>type=str</i>)
<code>__revision__</code>	Value: '\$Revision: 1046 \$' (<i>type=str</i>)

24 Module networkx.queues

Helper queues for use in graph searching.

LIFO: Last in first out queue (stack)

FIFO: First in first out queue

Priority(fcn): Priority queue with items are sorted by fcn

Random: Random queue

`q.append(item)` -- add an item to the queue

`q.extend(items)` -- equivalent to: `for item in items: q.append(item)`

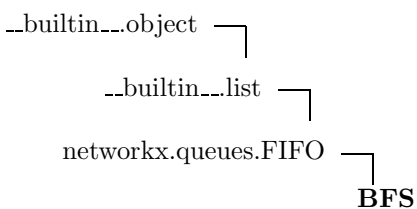
`q.pop()` -- return the top item from the queue

`len(q)` -- number of items in `q` (also `q.__len__()`)

24.1 Variables

Name	Description
<code>__author__</code>	Value: 'Aric Hagberg (hagberg@lanl.gov)' (<i>type=str</i>)
<code>__credits__</code>	Value: '' (<i>type=str</i>)
<code>__date__</code>	Value: '\$Date: 2005-03-30 16:56:28 -0700 (Wed, 30 Mar - 2005) \$' (<i>type=str</i>)
<code>__revision__</code>	Value: '\$Revision: 911 \$' (<i>type=str</i>)

24.2 Class BFS



Breadth first search queue

24.2.1 Methods

__init__(self) Overrides: <code>networkx.queues.FIFO.__init__</code>

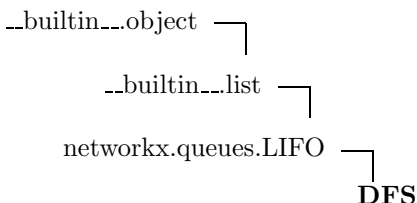
update(*self*, *item*)

Inherited from list: `__add__`, `__contains__`, `__delitem__`, `__delslice__`, `__eq__`, `__ge__`, `__getattr__`, `__getitem__`, `__getslice__`, `__gt__`, `__hash__`, `__iadd__`, `__imul__`, `__iter__`, `__le__`, `__len__`, `__lt__`, `__mul__`, `__ne__`, `__new__`, `__repr__`, `__reversed__`, `__rmul__`, `__setitem__`, `__setslice__`, `append`, `count`, `extend`, `index`, `insert`, `remove`, `reverse`, `sort`

Inherited from object: `__delattr__`, `__reduce__`, `__reduce_ex__`, `__setattr__`, `__str__`

Inherited from FIFO: `pop`

24.3 Class `DFS`



Depth first search queue

24.3.1 Methods

__init__(*self*)

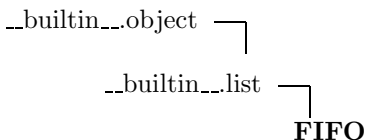
Overrides: `networkx.queues.LIFO.__init__`

update(*self*, *item*)

Inherited from list: `__add__`, `__contains__`, `__delitem__`, `__delslice__`, `__eq__`, `__ge__`, `__getattr__`, `__getitem__`, `__getslice__`, `__gt__`, `__hash__`, `__iadd__`, `__imul__`, `__iter__`, `__le__`, `__len__`, `__lt__`, `__mul__`, `__ne__`, `__new__`, `__repr__`, `__reversed__`, `__rmul__`, `__setitem__`, `__setslice__`, `append`, `count`, `extend`, `index`, `insert`, `pop`, `remove`, `reverse`, `sort`

Inherited from object: `__delattr__`, `__reduce__`, `__reduce_ex__`, `__setattr__`, `__str__`

24.4 Class `FIFO`



Known Subclasses: `BFS`

24.4.1 Methods

__init__(*self*)

Overrides: `__builtin__.list.__init__`

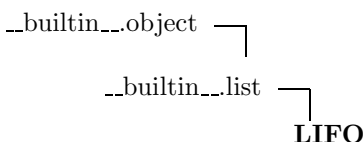
pop(*self*)

Overrides: `__builtin__.list.pop`

Inherited from list: `__add__`, `__contains__`, `__delitem__`, `__delslice__`, `__eq__`, `__ge__`, `__getattr__`, `__getitem__`, `__getslice__`, `__gt__`, `__hash__`, `__iadd__`, `__imul__`, `__iter__`, `__le__`, `__len__`, `__lt__`, `__mul__`, `__ne__`, `__new__`, `__repr__`, `__reversed__`, `__rmul__`, `__setitem__`, `__setslice__`, `append`, `count`, `extend`, `index`, `insert`, `remove`, `reverse`, `sort`

Inherited from object: `__delattr__`, `__reduce__`, `__reduce_ex__`, `__setattr__`, `__str__`

24.5 Class LIFO



Known Subclasses: DFS

24.5.1 Methods

__init__(*self*)

Overrides: `__builtin__.list.__init__`

Inherited from list: `__add__`, `__contains__`, `__delitem__`, `__delslice__`, `__eq__`, `__ge__`, `__getattr__`, `__getitem__`, `__getslice__`, `__gt__`, `__hash__`, `__iadd__`, `__imul__`, `__iter__`, `__le__`, `__len__`, `__lt__`, `__mul__`, `__ne__`, `__new__`, `__repr__`, `__reversed__`, `__rmul__`, `__setitem__`, `__setslice__`, `append`, `count`, `extend`, `index`, `insert`, `pop`, `remove`, `reverse`, `sort`

Inherited from object: `__delattr__`, `__reduce__`, `__reduce_ex__`, `__setattr__`, `__str__`

24.6 Class Priority

24.6.1 Methods

__init__(*self*, *f*=<function <lambda> at 0xb7c428ec>)

__len__(*self*)

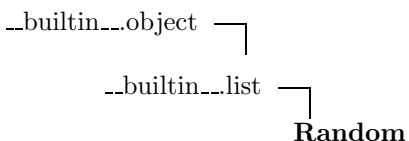
append(*self*, *item*)

extend(*self*, *items*)

pop(*self*)

smallest(*self*)

24.7 Class Random



Known Subclasses: RFS

24.7.1 Methods

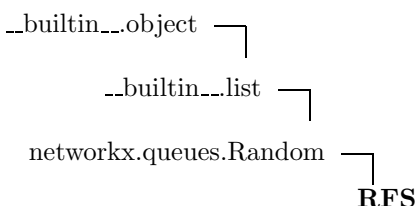
<code>__init__(self)</code> Overrides: <code>__builtin__.list.__init__</code>

<code>pop(self)</code> Overrides: <code>__builtin__.list.pop</code>

Inherited from list: `__add__`, `__contains__`, `__delitem__`, `__delslice__`, `__eq__`, `__ge__`, `__getattr__`, `__getitem__`, `__getslice__`, `__gt__`, `__hash__`, `__iadd__`, `__imul__`, `__iter__`, `__le__`, `__len__`, `__lt__`, `__mul__`, `__ne__`, `__new__`, `__repr__`, `__reversed__`, `__rmul__`, `__setitem__`, `__setslice__`, `append`, `count`, `extend`, `index`, `insert`, `remove`, `reverse`, `sort`

Inherited from object: `__delattr__`, `__reduce__`, `__reduce_ex__`, `__setattr__`, `__str__`

24.8 Class RFS



Random search queue

24.8.1 Methods

<code>__init__(self)</code> Overrides: <code>networkx.queues.Random.__init__</code>

<code>update(self, item)</code>

Inherited from list: `__add__`, `__contains__`, `__delitem__`, `__delslice__`, `__eq__`, `__ge__`, `__getattr__`, `__getitem__`, `__getslice__`, `__gt__`, `__hash__`, `__iadd__`, `__imul__`, `__iter__`, `__le__`, `__len__`, `__lt__`, `__mul__`, `__ne__`, `__new__`, `__repr__`, `__reversed__`, `__rmul__`, `__setitem__`, `__setslice__`, `append`, `count`, `extend`, `index`, `insert`, `remove`, `reverse`, `sort`

Inherited from object: `__delattr__`, `__reduce__`, `__reduce_ex__`, `__setattr__`, `__str__`

Inherited from Random: `pop`

25 Module `networkx.release`

Release data for NetworkX.

25.1 Variables

Name	Description
authors	Value: {'Swart': ('Pieter Swart', 'swart@lanl.gov'), '-Schult': ('Dan Schult', 'dschu... (<i>type=dict</i>)
date	Value: 'Sun Aug 21 08:07:12 2005' (<i>type=str</i>)
description	Value: 'A package for creating and manipulating large -graphs and networks.' (<i>type=str</i>)
keywords	Value: ['Networks', 'Graph Theory', 'Mathematics'] (<i>type=list</i>)
license	Value: 'LGPL' (<i>type=str</i>)
long_description	Value: '\nNetworkX is a python package for the creatio-n, manipulation, and\nstudy of... (<i>type=str</i>)
name	Value: 'networkx' (<i>type=str</i>)
platforms	Value: ['Linux', 'Mac OSX', 'Windows XP/2000/NT'] (<i>type=list</i>)
url	Value: 'http://networkx.sourceforge.net' (<i>type=str</i>)
version	Value: '0.24' (<i>type=str</i>)

26 Module `networkx.search`

Search algorithms, shortest path, spanning trees, etc.

See also `networkx.paths`.

The following search methods available, see the documentation below.

`number_connected_components(G)`

`connected_components(G)`

`connected_component_subgraphs(G)`

`dfs_preorder(G, v=None)`

`dfs_postorder(G, v=None)`

`dfs_predecessor(G, v=None)`

`dfs_successor(G, v=None)`

`bfs_length(G, source=None, target=None)`

`bfs_path(G, source, target=None)`

`dfs_forest(G, v=None)`

These algorithms are based on Program 18.10 "Generalized graph search", page 128, Algorithms in C, Part 5, Graph Algorithms by Robert Sedgewick

Reference:

```
@Book{sedgewick-2001-algorithms-5,  
author =      {Robert Sedgewick},  
title =       {Algorithms in C, Part 5: Graph Algorithms},  
publisher =    {Addison Wesley Professional},  
year =        {2001},  
edition =     {3rd},  
}
```

26.1 Functions

<code>bfs_length(G, source, target=None)</code>
Return a dictionary of nodes with the shortest path length from source.

`bfs_path(G, source, target=None)`

Return a dictionary of nodes with the paths
from source to all reachable nodes.
Optional `target=target` produces only one path as a list.

`connected_component_subgraphs(G)`

Return a list of graphs of each connected component of *G*.
The list is ordered from largest connected component to smallest.
To get the largest connected component:

```
>>> H=connected_component_subgraphs(G)[0]
```

`connected_components(G)`

Return a list of lists of nodes in each connected component of *G*.
The list is ordered from largest connected component to smallest.

`dfs_forest(G, v=None)`

Return a forest of trees built from depth first search (DFS).
Optional `v=v` limits search to component of graph containing *v*
and will return a single tree.

`dfs_postorder(G, v=None)`

Return a list of nodes ordered by depth first search (DFS) postorder.
If the graph has more than one component return a list of lists.
Optional `v=v` limits search to component of graph containing *v*.

`dfs_predecessor(G, v=None)`

Return a dictionary of nodes each with a list of predecessor
nodes in depth first search (DFS) order.
Optional `v=v` limits search to component of graph containing *v*.

`dfs_preorder(G, v=None)`

Return a list of nodes ordered by depth first search (DFS) preorder.
If the graph has more than one component return a list of lists.
Optional `v=v` limits search to component of graph containing *v*.

`dfs_successor(G, v=None)`

Return a dictionary of nodes each with a list of successor
nodes in depth first search (DFS) order.
Optional `v=v` limits search to component of graph containing *v*.

<code>is_connected(<i>G</i>)</code>

True if <i>G</i> is connected

<code>node_connected_component(<i>G</i>, <i>v</i>)</code>

Return the connected component to which <i>v</i> belongs as a list of nodes.

<code>number_connected_components(<i>G</i>)</code>

Return number of connected components of <i>G</i> .

26.2 Variables

Name	Description
<code>__author__</code>	Value: 'Aric Hagberg (hagberg@lanl.gov)\nDan Schult(ds-chult@colgate.edu)' (<i>type=str</i>)
<code>__credits__</code>	Value: '' (<i>type=str</i>)
<code>__date__</code>	Value: '\$Date: 2005-06-15 08:19:25 -0600 (Wed, 15 Jun - 2005) \$' (<i>type=str</i>)
<code>__revision__</code>	Value: '\$Revision: 1026 \$' (<i>type=str</i>)

27 Module `networkx.search_class`

Graph search classes

The search algorithms are implemented as an abstract class with visitor functions that are called at points during the algorithm. By designing different visitor functions the search algorithms can produce shortest path lengths, forests of search trees, etc.

The simplest way to access the search algorithms is by using predefined visitor classes and search functions. See the module `networkx.search`.

These algorithms are based on Program 18.10 "Generalized graph search", page 128, Algorithms in C, Part 5, Graph Algorithms by Robert Sedgewick

Reference:

```
@Book{sedgewick-2001-algorithms-5,
author =      {Robert Sedgewick},
title =       {Algorithms in C, Part 5: Graph Algorithms},
publisher =   {Addison Wesley Professional},
year =        {2001},
edition =     {3rd},
}
```

27.1 Variables

Name	Description
<code>__author__</code>	Value: 'Aric Hagberg (hagberg@lanl.gov)' (<i>type=</i> <code>str</code>)
<code>__credits__</code>	Value: '' (<i>type=</i> <code>str</code>)
<code>__date__</code>	Value: '\$Date: 2005-06-15 08:17:35 -0600 (Wed, 15 Jun - 2005) \$' (<i>type=</i> <code>str</code>)
<code>__revision__</code>	Value: '\$Revision: 1025 \$' (<i>type=</i> <code>str</code>)

27.2 Class `Forest`

```
__builtin__.object └─
networkx.search_class.Search └─
                                Forest
```

Forest visitor: build a forest of trees as a list of `networkx` DiGraphs.

27.2.1 Methods

```
__init__(self, G, queue=<class 'networkx.queues.DFS'>, **kws)
```

Overrides: `networkx.search_class.Search.__init__`

```
end_tree(self, v)
```

Visitor function called at the search end of each connected component.

Overrides: `networkx.search_class.Search.end_tree`

```
lastseen_edge(self, e)
```

Visitor function called the last time an edge is encountered.

Overrides: `networkx.search_class.Search.lastseen_edge`

```
start_tree(self, v)
```

Visitor function called at the search start of each connected component.

Overrides: `networkx.search_class.Search.start_tree`

Inherited from object: `__delattr__`, `__getattr__`, `__hash__`, `__new__`, `__reduce__`, `__reduce_ex__`, `__repr__`, `__setattr__`, `__str__`

Inherited from Search: `firstseen_edge`, `firstseen_vertex`, `lastseen_vertex`, `search`

27.3 Class Length

```
__builtin__.object
```

```
networkx.search_class.Search
```

Length

Path length visitor.

Returns dictionary of path lengths from vertex `v`.

Useful especially in BFS (gives shortest paths).

27.3.1 Methods

```
__init__(self, G, queue=<class 'networkx.queues.BFS'>, **kws)
```

Overrides: `networkx.search_class.Search.__init__`

```
lastseen_edge(self, e)
```

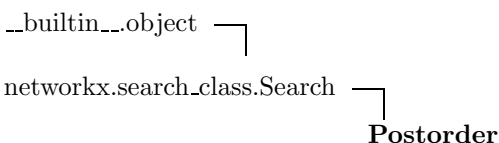
Visitor function called the first time an edge is encountered.

Overrides: `networkx.search_class.Search.lastseen_edge`

Inherited from object: `__delattr__`, `__getattr__`, `__hash__`, `__new__`, `__reduce__`, `__reduce_ex__`, `__repr__`, `__setattr__`, `__str__`

Inherited from Search: `end_tree`, `firstseen_edge`, `firstseen_vertex`, `lastseen_vertex`, `search`, `start_tree`

27.4 Class `Postorder`



Postorder visitor

Builds a list of nodes in postorder of search.

Returns a list of lists if the graph is not connected.

27.4.1 Methods

<code>__init__(self, G, queue=<class 'networkx.queues.DFS'>, **kws)</code> Overrides: <code>networkx.search_class.Search.__init__</code>

<code>end_tree(self, v)</code>

Visitor function called at the search end of each connected component.

Overrides: <code>networkx.search_class.Search.end_tree</code>

<code>lastseen_vertex(self, v)</code>

Visitor function called the last time a vertex is encountered.

Overrides: <code>networkx.search_class.Search.lastseen_vertex</code>

<code>start_tree(self, v)</code>

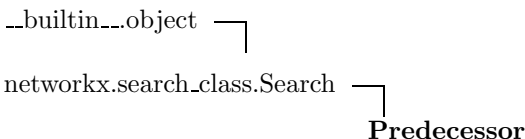
Visitor function called at the search start of each connected component.

Overrides: <code>networkx.search_class.Search.start_tree</code>

Inherited from object: `__delattr__`, `__getattr__`, `__hash__`, `__new__`, `__reduce__`, `__reduce_ex__`, `__repr__`, `__setattr__`, `__str__`

Inherited from Search: `firstseen_edge`, `firstseen_vertex`, `lastseen_edge`, `search`

27.5 Class Predecessor



Predecessor visitor

Builds a dict of nodes with successor vertex list as data.

path method returns path lengths from source to target.

27.5.1 Methods

`__init__(self, G, queue=<class 'networkx.queues.DFS'>, **kws)`

Overrides: `networkx.search_class.Search.__init__`

`firstseen_vertex(self, v)`

Visitor function called the first time a vertex is encountered.

Overrides: `networkx.search_class.Search.firstseen_vertex`

`lastseen_edge(self, e)`

Visitor function called the last time an edge is encountered.

Overrides: `networkx.search_class.Search.lastseen_edge`

`path(self, target)`

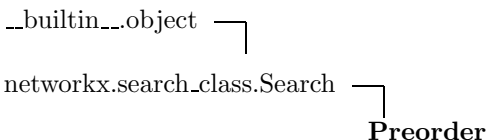
Gets one shortest path to target out of predecessor hash.

There might be more than one

Inherited from object: `__delattr__`, `__getattr__`, `__hash__`, `__new__`, `__reduce__`, `__reduce_ex__`, `__repr__`, `__setattr__`, `__str__`

Inherited from Search: `end_tree`, `firstseen_edge`, `lastseen_vertex`, `search`, `start_tree`

27.6 Class Preorder



Preorder visitor

Builds a list of nodes in preorder of search.

Returns a list of lists if the graph is not connected.

27.6.1 Methods

```
__init__(self, G, queue=<class 'networkx.queues.DFS'>, **kws)
```

Overrides: `networkx.search_class.Search.__init__`

```
end_tree(self, v)
```

Visitor function called at the search end of each connected component.

Overrides: `networkx.search_class.Search.end_tree`

```
firstseen_vertex(self, v)
```

Visitor function called the first time a vertex is encountered.

Overrides: `networkx.search_class.Search.firstseen_vertex`

```
start_tree(self, v)
```

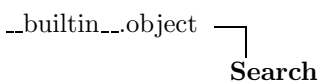
Visitor function called at the search start of each connected component.

Overrides: `networkx.search_class.Search.start_tree`

Inherited from object: `__delattr__`, `__getattr__`, `__hash__`, `__new__`, `__reduce__`, `__reduce_ex__`, `__repr__`, `__setattr__`, `__str__`

Inherited from Search: `firstseen_edge`, `lastseen_edge`, `lastseen_vertex`, `search`

27.7 Class Search



Known Subclasses: `Forest`, `Length`, `Postorder`, `Predecessor`, `Preorder`, `Successor`

Generic graph traversal (`search`) class.

Users should generally use the search functions defined below.

e.g. to get a list of all nodes of `G` in breadth first search (BFS) order from `v` use

```
vertex_list=bfs_preorder(G,v)
```

To search the graph `G` from `v` do the following:

```
S=Search(G,queue=DFS)
S.search(v=v)
```

Depending on the type of queue you will get a different traversal type.

You may use any of the following queues from the `Queues` class:

Name	Queue	Traversal
DFS	LIFO	Depth First Search
BFS	FIFO	Breadth First Search
Random	Random	Random search

The generic search produces no data and thus is of limited utility. Visitor callback functions are called at points along the search which may be used to store shortest path data

27.7.1 Methods

```
__init__(self, G, queue=<class 'networkx.queues.DFS'>)
Overrides: __builtin__.object.__init__
```

```
end_tree(self, v)
```

Visitor function called at the search end of each connected component.

```
firstseen_edge(self, e)
```

Visitor function called the first time an edge is encountered.

```
firstseen_vertex(self, v)
```

Visitor function called the first time a vertex is encountered.

```
lastseen_edge(self, e)
```

Visitor function called the last time an edge is encountered.

```
lastseen_vertex(self, v)
```

Visitor function called the last time a vertex is encountered.

`search(self, v=None)`

Search the graph.

The search method is deteremined by the initialization of the search object.

The optional `v=` argument can be a single vertex a list or `None`.

`v=v:` search the component of `G` reachable from `v`
`v=vlist:` search the component of `G` reachable from `v`
`v=None:` search the entire graph `G` even if it isn't connected

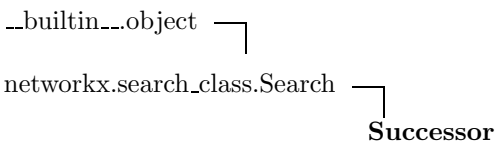
Call visitor functions along the way.

`start_tree(self, v)`

Visitor function called at the search start
of each connected component.

Inherited from object: `__delattr__`, `__getattr__`, `__hash__`, `__new__`, `__reduce__`, `__reduce_ex__`, `__repr__`, `__setattr__`, `__str__`

27.8 Class Successor



Successor visitor

Builds a dict of nodes with sucessor vertex list as data.

27.8.1 Methods

`__init__(self, G, queue=<class 'networkx.queueues.DFS'>, **kws)`

Overrides: `networkx.search_class.Search.__init__`

`firstseen_vertex(self, v)`

Visitor function called the first time a vertex is encountered.

Overrides: `networkx.search_class.Search.firstseen_vertex`

`lastseen_edge(self, e)`

Visitor function called the last time an edge is encountered.

Overrides: `networkx.search_class.Search.lastseen_edge`

Inherited from object: `__delattr__`, `__getattr__`, `__hash__`, `__new__`, `__reduce__`, `__reduce_ex__`, `__repr__`, `__setattr__`, `__str__`

Inherited from Search: `end_tree`, `firstseen_edge`, `lastseen_vertex`, `search`, `start_tree`

28 Module `networkx.spectrum`

Laplacian, adjacency matrix, and spectrum of graphs.

Uses `Numeric`.

28.1 Functions

`adj_matrix(G, u=None)`

Return adjacency matrix of graph
If *u* is defined return row of adjacency matrix at row *u*.

`generalized_laplacian(G)`

Return generalized Laplacian of graph

See Spectral Graph Theory by Fan Chung-Graham.

`laplacian(G)`

Return standard Laplacian of graph

28.2 Variables

Name	Description
<code>__author__</code>	Value: 'Aric Hagberg (hagberg@lanl.gov)\nPieter Swart - (swart@lanl.gov)\nDan Schult(d... (<i>type=</i> <code>str</code>)
<code>__credits__</code>	Value: '' (<i>type=</i> <code>str</code>)
<code>__date__</code>	Value: '\$Date: 2005-06-15 14:18:07 -0600 (Wed, 15 Jun - 2005) \$\n' (<i>type=</i> <code>str</code>)
<code>__revision__</code>	Value: '\$Revision: 1044 \$\n' (<i>type=</i> <code>str</code>)

29 Module `networkx.threshold`

Threshold Graphs - Creation, manipulation and identification.

29.1 Functions

`betweenness_sequence`(*creation_sequence*)

Return betweenness for the threshold graph with the given creation sequence. The result is unscaled. To scale the values to the interval `[0,1]` divide by $(n-1)*(n-2)$.

`cluster_sequence`(*creation_sequence*)

Return cluster sequence for the given threshold graph creation sequence.

`creation_sequence`(*degree_sequence*, *with_labels*=False, *compact*=False)

Determines the creation sequence for the given threshold degree sequence.

The creation sequence is a list of single characters 'd' or 'i': 'd' for dominating or 'i' for isolated vertices. Dominating vertices are connected to all vertices present when it is added. The first node added is by convention 'd'.

If `with_labels==True`:

Returns a list of 2-tuples containing the vertex number and a character 'd' or 'i' which describes the type of vertex.

If `compact==True`:

Returns the creation sequence in a compact form that is the number of 'i's and 'd's alternating.

Examples:

`[1,2,2,3]` represents d,i,i,d,d,i,i,i

`[3,1,2]` represents d,d,d,i,d,d

Notice that the first number is the first vertex to be used for construction and so is always 'd'.

`with_labels` and `compact` cannot both be True.

Returns None if the sequence is not a threshold sequence

creation_sequence_to_weights(*creation_sequence*)

Returns a list of node weights which create the threshold graph designated by the creation sequence. The weights are scaled so that the threshold is 1.0. The order of the nodes is the same as that in the creation sequence.

degree_correlation(*creation_sequence*)

Return the degree-degree correlation over all edges.

degree_sequence(*creation_sequence*)

Return degree sequence for the threshold graph with the given creation sequence

density(*creation_sequence*)

Return the density of the graph with this creation_sequence. The density is the fraction of possible edges present.

eigenvalues(*creation_sequence*)

Return sequence of eigenvalues of the Laplacian of the threshold graph for the given creation_sequence.

Based on the Ferrer's diagram method. The spectrum is integral and is the conjugate of the degree sequence.

See:

```
@Article{degree-merris-1994,
  author = {Russel Merris},
  title = {Degree maximal graphs are Laplacian integral},
  journal = {Linear Algebra Appl.},
  year = {1994},
  volume = {199},
  pages = {381--389},
}
```

find_alternating_4_cycle(*G*)

Returns False if there aren't any alternating 4 cycles. Otherwise returns the cycle as [a,b,c,d] where (a,b) and (c,d) are edges and (a,c) and (b,d) are not.

find_creation_sequence(*G*)

Find a threshold subgraph that is close to largest in *G*.
Returns the labeled creation sequence of that threshold graph.

find_threshold_graph(*G*)

Return a threshold subgraph that is close to largest in *G*.
The threshold graph will contain the largest degree node in *G*.

If you just want the creation sequence you can use
`creation_sequence(find_threshold_graph(G).degree(with_labels=True),
labeled=True)`

is_threshold_graph(*G*)

Returns True if *G* is a threshold graph.

is_threshold_sequence(*degree_sequence*)

Returns True if the sequence is a threshold degree sequence.

Uses the property that a threshold graph must be constructed by adding either dominating or isolated nodes. Thus, it can be deconstructed iteratively by removing a node of degree zero or a node that connects to the remaining nodes. If this deconstruction fails then the sequence is not a threshold sequence.

left_d_threshold_sequence(*n, m*)

Create a skewed threshold graph with a given number of vertices (*n*) and a given number of edges (*m*).

The routine returns an unlabeled creation sequence for the threshold graph.

FIXME: describe algorithm

make_compact(*creation_sequence*)

Returns the creation sequence in a compact form that is the number of 'i's and 'd's alternating.

Examples:

[1,2,2,3] represents d,i,i,d,d,i,i,i.

[3,1,2] represents d,d,d,i,d,d.

Notice that the first number is the first vertex to be used for construction and so is always 'd'.

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

Create a random threshold sequence of size *n*.
 A creation sequence is built by randomly choosing d's with probability *p* and i's with probability 1-*p*.

```
>>> s=random_threshold_sequence(10,0.5)
```

returns a threshold sequence of length 10 with equal probability of an i or a d at each position.

A "random" threshold graph can be built with

```
>>> G=threshold_graph(random_threshold_sequence(n,p))
```

right_d_threshold_sequence(*n, m*)

Create a skewed threshold graph with a given number of vertices (*n*) and a given number of edges (*m*).

The routine returns an unlabeled creation sequence for the threshold graph.

FIXME: describe algorithm

shortest_path(*creation_sequence, u, v*)

Find the shortest path between *u* and *v* in a threshold graph *G* with the given *creation_sequence*.

For an unlabeled *creation_sequence*, the vertices *u* and *v* must be integers in (0,len(sequence)) referring to the position of the desired vertices in the sequence.

For a labeled *creation_sequence*, *u* and *v* are labels of vertices.

Use *cs=creation_sequence(degree_sequence,with_labels=True)* to convert a degree sequence to a creation sequence.

Returns a list of vertices from *u* to *v*.

Example: if they are neighbors, it returns [*u,v*]

shortest_path_length(*creation_sequence*, *i*)

Return the shortest path length from indicated node to every other node for the threshold graph with the given creation sequence.

Node is indicated by index *i* in *creation_sequence* unless *creation_sequence* is labeled in which case, *i* is taken to be the label of the node.

Paths lengths in threshold graphs are at most 2.
Length to unreachable nodes is set to -1.

swap_d(*cs*, *p_split*=1.0, *p_combine*=1.0, *seed*=None)

Perform a "swap" operation on a threshold sequence.

The swap preserves the number of nodes and edges in the graph for the given sequence.
The resulting sequence is still a threshold sequence.

Perform one split and one combine operation on the 'd's of a creation sequence for a threshold graph.
This operation maintains the number of nodes and edges in the graph, but shifts the edges from node to node maintaining the threshold quality of the graph.

threshold_graph(*creation_sequence*)

Create a threshold graph from the creation sequence or compact creation sequence.

The input sequence can be a

creation sequence (e.g. ['d','i','d','d','d','i'])
labeled creation sequence (e.g. [(0,'d'),(2,'d'),(1,'i')])
compact creation sequence (e.g. [2,1,1,2,0])

Use `cs=creation_sequence(degree_sequence,labeled=True)` to convert a degree sequence to a creation sequence.

Returns None if the sequence is not valid

triangle_sequence(*creation_sequence*)

Return triangle sequence for the given threshold graph creation sequence.

triangles(*creation_sequence*)

Compute number of triangles in the threshold graph with the given creation sequence.

uncompact(*compact_creation_sequence*)

Converts a compact creation sequence for a threshold graph to a standard creation sequence (unlabeled). See `creation_sequence`.

weights_to_creation_sequence(*weights*, *threshold=1*, *with_labels=False*, *compact=False*)

Returns a creation sequence for a threshold graph determined by the weights and threshold given as input. If the sum of two node weights is greater than the threshold value, an edge is created between these nodes.

The creation sequence is a list of single characters 'd' or 'i': 'd' for dominating or 'i' for isolated vertices. Dominating vertices are connected to all vertices present when it is added. The first node added is by convention 'd'.

If `with_labels==True`:

Returns a list of 2-tuples containing the vertex number and a character 'd' or 'i' which describes the type of vertex.

If `compact==True`:

Returns the creation sequence in a compact form that is the number of 'i's and 'd's alternating.

Examples:

[1,2,2,3] represents d,i,i,d,d,i,i,i

[3,1,2] represents d,d,d,i,d,d

Notice that the first number is the first vertex to be used for construction and so is always 'd'.

`with_labels` and `compact` cannot both be `True`.

29.2 Variables

Name	Description
<code>__author__</code>	Value: 'Aric Hagberg (hagberg@lanl.gov)\nPieter Swart - (swart@lanl.gov)\nDan Schult (... (<i>type=str</i>)
<code>__credits__</code>	Value: '' (<i>type=str</i>)

continued on next page

Name	Description
<code>__date__</code>	Value: '\$Date: 2005-06-17 08:06:22 -0600 (Fri, 17 Jun - 2005) \$' (<i>type=str</i>)
<code>__version__</code>	Value: '\$Revision: 1049 \$' (<i>type=str</i>)

30 Module `networkx.utils`

Utilities for `networkx` package

30.1 Functions

`discrete_sequence(n, **kws)`

Return sample sequence of length `n` from a given discrete distribution

distribution=histogram of values, will be normalized

`gsl_pareto_sequence(n, **kws)`

Return sample sequence of length `n` from a Pareto distribution.

`gsl_poisson_sequence(n, **kws)`

Return sample sequence of length `n` from a Poisson distribution.

`gsl_powerlaw_sequence(n, **kws)`

Return sample sequence of length `n` from a power law distribution.

`gsl_uniform_sequence(n, **kws)`

Return sample sequence of length `n` from a uniform distribution.

`is_list_of_ints(intlist)`

Return True if list is a list of ints.

`is_singleton(obj)`

Is string-like or not iterable.

`is_string_like(obj)`

Check if `obj` is string.

`iterable(obj)`

Return True if `obj` is iterable with a well-defined `len()`

`pareto_sequence(n, **kws)`

Return sample sequence of length `n` from a Pareto distribution.

powerlaw_sequence(*n*, ****kws**)

Return sample sequence of length *n* from a power law distribution.

scipy_discrete_sequence(*n*, ****kws**)

Return sample sequence of length *n* from a given discrete distribution

distribution=histogram of values, will be normalized

scipy_pareto_sequence(*n*, ****kws**)

Return sample sequence of length *n* from a Pareto distribution.

scipy_poisson_sequence(*n*, ****kws**)

Return sample sequence of length *n* from a Poisson distribution.

scipy_powerlaw_sequence(*n*, ****kws**)

Return sample sequence of length *n* from a power law distribution.

scipy_uniform_sequence(*n*)

Return sample sequence of length *n* from a uniform distribution.

uniform_sequence(*n*)

Return sample sequence of length *n* from a uniform distribution.

30.2 Variables

Name	Description
<code>__author__</code>	Value: 'Aric Hagberg (hagberg@lanl.gov)\nDan Schult(ds-chult@colgate.edu)' (<i>type=</i> str)
<code>__credits__</code>	Value: '' (<i>type=</i> str)
<code>__date__</code>	Value: '\$Date: 2005-06-15 08:30:40 -0600 (Wed, 15 Jun - 2005) \$' (<i>type=</i> str)
<code>__revision__</code>	Value: '\$Revision: 1029 \$' (<i>type=</i> str)

31 Module `networkx.xbase`

Methods for general graphs (`XGraph`) and digraphs (`XDiGraph`) allowing self-loops, multiple edges, arbitrary (hashable) objects as nodes and arbitrary objects associated with edges.

The `XGraph` and `XDiGraph` classes are extensions of the `Graph` and `DiGraph` classes in `base.py`. The key difference is that an `XGraph` edge is a 3-tuple `e=(n1,n2,x)`, representing an undirected edge between nodes `n1` and `n2` that is decorated with the object `x`. Here `n1` and `n2` are (hashable) node objects and `x` is a (not necessarily hashable) edge object. Since the edge is undirected, edge `(n1,n2,x)` is equivalent to edge `(n2,n1,x)`.

An `XDiGraph` edge is a similar 3-tuple `e=(n1,n2,x)`, with the additional property of directedness. I.e. `e=(n1,n2,x)` is a directed edge from `n1` to `n2` decorated with the object `x`, and is not equivalent to the edge `(n2,n1,x)`.

Whether a graph or digraph allow self-loops or multiple edges is determined at the time of object instantiation via specifying the parameters `selfloops=True/False` and `multiedges=True/False`. For example,

an empty `XGraph` is created with:

```
>>> G=XGraph()
```

which is equivalent to

```
>>> G=XGraph(name="No Name", selfloops=False, multiedges=False)
```

and similarly for `XDiGraph`.

```
>>> G=XDiGraph(name="empty", multiedges=True)
```

creates an empty digraph `G` with `G.name="empty"`, that do not allow the addition of selfloops but do allow for multiple edges.

`XGraph` and `XDiGraph` are implemented using a data structure based on an adjacency list implemented as a dictionary of dictionaries. The outer dictionary is keyed by node to an inner dictionary keyed by neighboring nodes to the edge data/labels/objects (which default to 1 to correspond the datastructure used in classes `Graph` and `DiGraph`). If `multiedges=True`, a list of edge data/labels/objects is stored as the value of the inner dictionary. This double dict structure mimics a sparse matrix and allows fast addition, deletion and lookup of nodes and neighbors in large graphs. The underlying datastructure should only be visible in this module. In all other modules,

graph-like objects are manipulated solely via the methods defined here and not by acting directly on the datastructure.

Similarities between `XGraph` and `Graph`

`XGraph` and `Graph` differ fundamentally; `XGraph` edges are 3-tuples $(n1, n2, x)$ and `Graph` edges are 2-tuples $(n1, n2)$. `XGraph` inherits from the `Graph` class, and `XDiGraph` from the `DiGraph` class.

They do share important similarities.

1. Edgeless graphs are the same in `XGraph` and `Graph`.

For an edgeless graph, represented by `G` (member of the `Graph` class) and `XG` (member of `XGraph` class), there is no difference between the datastructures `G.adj` and `XG.adj`, other than in the ordering of the keys in the `adj` dict.

2. Basic graph construction code for `G=Graph()` will also work for `G=XGraph()`. In the `Graph` class, the simplest graph construction consists of a graph creation command `G=Graph()` followed by a list of graph construction commands, consisting of successive calls to the methods:

```
G.add_node, G.add_nodes_from, G.add_edge, G.add_edges, G.add_path,  
G.add_cycle, G.delete_node, G.delete_nodes_from, G.delete_edge,  
G.delete_edges_from
```

with all edges specified as 2-tuples,

If one replaces the graph creation command with `G=XGraph()`, and then apply the identical list of construction commands, the resulting `XGraph` object will be a simple graph `G` with identical datastructure `G.adj`. This property ensures reuse of code developed for graph generation in the `Graph` class.

Notation

The following shorthand is used throughout NetworkX documentation and code: (we use mathematical notation n, v, w, \dots to indicate a node, v =vertex=node).

`G, G1, G2, H, etc:`

Graphs

`n, n1, n2, u, v, v1, v2:`

nodes (vertices)

`nlist:`

a list of nodes (vertices)

nbunch:

a "bunch" of nodes (vertices).

an nbunch is any iterable (non-string) container of nodes that is not itself a node of the graph.

e=(n1,n2):

an edge (a python "2-tuple"), also written n1-n2 (if undirected) and n1->n2 (if directed). In Xgraph G.add_edge(n1,n2) is equivalent to add_edge(n1,n2,1). However, G.delete_edge(n1,n2) will delete all edges between n1 and n2.

e=(n1,n2,x):

an edge triple ("3-tuple") containing the two nodes connected and the edge data/label/object stored associated with the edge. The object x, or a list of objects (if multiedges=True), can be obtained using G.get_edge(n1,n2)

elist:

a list of edges (as 2- or 3-tuples)

ebunch:

a bunch of edges (as 2- or 3-tuples)

an ebunch is any iterable (non-string) container of edge-tuples (either 2-tuples, 3-tuples or a mixture). (similar to nbunch, also see add_edge).

Warning:

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

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

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

Methods

The XGraph class provides rudimentary graph operations:

Mutating Graph methods

`G.add_node(n)`, `G.add_nodes_from(nbunch)`
`G.delete_node(n)`, `G.delete_nodes_from(nbunch)`
`G.add_edge(n1,n2,x)`, `G.add_edge(e)`,
`G.add_edges_from(ebunch)`
`G.delete_edge(n1,n2)`, `G.delete_edge(n1,n2,x)`, `G.delete_edge(e)`,
`G.delete_edges_from(ebunch)`
`G.add_path(nlist)`
`G.add_cycle(nlist)`
`G.to_directed()`
`G.ban_multiedges()`
`G.allow_multiedges()`
`G.delete_multiedges()`
`G.ban_selfloops()`
`G.allow_selfloops()`
`G.delete_selfloops()`
`G.clear()`
`G.subgraph(nbunch, inplace=True)`

Non-mutating Graph methods

`G.has_node(n)`
`G.nodes()`
`G.nodes_iter()`
`G.order()`
`G.neighbors(n)`, `G.neighbors_iter(n)`

```
G.has_edge(n1,n2), G.has_neighbor(n1,n2)

G.edges(), G.edges(nbunch)
G.edges_iter(), G.edges_iter(nbunch,

G.size()

G.get_edge(n1,n2)

G.degree(), G.degree(n), G.degree(nbunch)

G.degree_iter(), G.degree_iter(n), G.degree_iter(nbunch)

G.number_of_selfloops()

G.nodes_with_selfloops()

G.selfloop_edges()

G.copy()

G.subgraph(nbunch)
```

Examples

Create an empty graph structure (a "null graph") with zero nodes and zero edges

```
>>> from networkx import *
>>> G=XGraph(directed=True) # default no-loops, no-multiedges
```

You can add nodes in the same way as the simple Graph class

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

You can add edges as for simple Graph class, but with optional edge data/labels/objects.

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

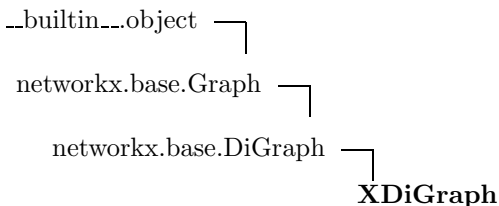
For graph coloring problems, one could use

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

31.1 Variables

Name	Description
<code>__author__</code>	Value: 'Aric Hagberg (hagberg@lanl.gov)\nPieter Swart - (swart@lanl.gov)\nDan Schult(d... (<i>type=</i> <code>str</code>)

31.2 Class `XDiGraph`



A class implementing general undirected digraphs, allowing (optional) self-loops, multiple edges, arbitrary (hashable) objects as nodes and arbitrary objects associated with edges.

As in `XGraph`, an `XDiGraph` edge is uniquely specified by a 3-tuple `e=(n1,n2,x)`, where `n1` and `n2` are (hashable) objects (nodes) and `x` is an arbitrary (and not necessarily unique) object associated with that edge.

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

`XDiGraph` replaces the following `DiGraph` methods:

`__init__`: read `multiedges` and `selfloops` kwds.

`add_edge`

`add_edges_from`

`delete_edge`

`delete_edges_from`

`has_edge`

`edges_iter`

`degree_iter`

`degree`

`copy`

`clear`

`subgraph`

`is_directed`

`to_directed`

XDiGraph also adds the following methods to those of DiGraph:

`allow_selfloops`

`remove_selfloops`

`ban_selfloops`

`allow_multiedges`

`remove_multiedges`

`ban_multiedges`

XDigraph adds the following methods to those of XGraph:

`has_successor`

`successors`

`successors_iter`

`has_predecessor`

`predecessors`

`predecessors_iter`

`out_degree`

`out_degree_iter`

`in_degree`

`in_degree_iter`

`to_undirected`

`is_directed`

31.2.1 Methods

```
__init__(self, **kwargs)
```

Initialize XDiGraph.

Optional arguments::

`name`: digraph name (default="No Name")

`selfloops`: if True then selfloops are allowed (default=False)

`multiedges`: if True then multiple edges are allowed (default=False)

Overrides: `networkx.base.DiGraph.__init__`

`add_edge(self, n1, n2=None, x=None)`

Add a single directed edge to the digraph.

Can be called as `G.add_edge(n1,n2,x)`
or as `G.add_edge(e)`, where `e=(n1,n2,x)`.

If called as `G.add_edge(n1,n2)` or `G.add_edge(e)`, with `e=(n1,n2)`, then this is interpreted as adding the edge `(n1,n2,1)`, so as to be compatible with the `Graph` and `DiGraph` classes.

`n1,n2` are (hashable) node objects, and are added silently to the `Graph` if not already present.

`x` is an arbitrary (not necessarily hashable) object associated with this edge. It can be used to associate one or more, labels, data records, weights or any arbitrary objects to edges.

For example, if the graph `G` was created with

```
>>> G=XDiGraph()
```

then `G.add_edge(1,2,"blue")` will add the directed edge `(1,2,"blue")`.

If `G.multiedges=False`, then a subsequent `G.add_edge(1,2,"red")` will change the above edge `(1,2,"blue")` into the edge `(1,2,"red")`.

On the other hand, if `G.multiedges=True`, then two successive calls to `G.add_edge(1,2,"red")` will result in 2 edges of the form `(1,2,"red")` that can not be distinguished from one another.

If `self.selfloops=False`, then any attempt to create a self-loop with `add_edge(n1,n1,x)` will have no effect on the digraph and will not elicit a warning.

Objects imbedded in the edges from `n1` to `n2` (if any), can be retrieved using `get_edge(n1,n2)`, or calling `edges(n1)` or `edge_iter(n1)` to return all edges attached to `n1`.

Overrides: `networkx.base.DiGraph.add_edge`

add_edges_from(*self*, *ebunch*)

Add multiple directed edges to the digraph.

ebunch: Container of edges. Each edge *e* in container will be added using `add_edge(e)`. See `add_edge` documentation.

The container must be iterable or an iterator.
It is iterated over once.

Overrides: `networkx.base.DiGraph.add_edges_from`

allow_multiedges(*self*)

Henceforth allow addition of multiedges (more than one edge between two nodes).

Warning: This causes all edge data to be converted to lists.

allow_selfloops(*self*)

Henceforth allow addition of self-loops
(edges from a node to itself).

This doesn't change the graph structure, only what you can do to it.

ban_multiedges(*self*)

Remove multiedges retaining the data from the first edge.
Henceforth do not allow multiedges.

ban_selfloops(*self*)

Remove self-loops from the graph and henceforth do not allow their creation.

copy(*self*)

Return a (shallow) copy of the digraph.

Return a new `XDiGraph` with same name and same attributes for selfloop and multiedges. Each node and each edge in original graph are added to the copy.

Overrides: `networkx.base.DiGraph.copy`

degree(*self*, *nbunch*=None, *with_labels*=False)

Return the out-degree of single node or of nbunch of nodes.
 If nbunch is omitted or nbunch=None, then return
 out-degrees of all nodes.

If with_labels=True, then return a dict that maps each n
 in nbunch to out_degree(n).

Any nodes in nbunch that are not in the graph are
 (quietly) ignored.

Overrides: networkx.base.DiGraph.degree

delete_edge(*self*, *n1*, *n2*=None, *x*=None)

Delete the directed edge (n1,n2,x) from the graph.

Can be called either as G.delete_edge(n1,n2,x)
 or as G.delete_edge(e), where e=(n1,n2,x).

If x is unspecified, i.e. if called with an edge e=(n1,n2),
 or as G.delete_edge(n1,n2), then delete all edges between n1 and n2.

If the edge does not exist, do nothing.

Overrides: networkx.base.DiGraph.delete_edge

delete_edges_from(*self*, *ebunch*, *data*=None)

Delete edges in ebunch from the graph.

ebunch: Container of edges. Each edge must be a 3-tuple
 (n1,n2,x) or a 2-tuple (n1,n2). The container must be
 iterable or an iterator, and is iterated over once. Edges
 that are not in the graph are ignored.

Overrides: networkx.base.DiGraph.delete_edges_from

delete_multiedges(*self*)

Remove multiedges retaining the data from the first edge

delete_selfloops(*self*)

Remove self-loops from the graph (edges from a node to itself).

`edges_iter(self, nbunch=None, with_labels=False)`

Return iterator that iterates once over each edge adjacent to nodes in `nbunch`, or over all edges in digraph if no nodes are specified.

See `add_node` for definition of `nbunch`.

Those nodes in `nbunch` that are not in the graph will be (quietly) ignored.

`with_labels=True` is not supported. (In that case you should probably use `neighbors()`.)

Overrides: `networkx.base.DiGraph.edges_iter`

`get_edge(self, n1, n2)`

Return the objects associated with each edge between `n1` and `n2`.

If `multiedges=False`, a single object is returned.

If `multiedges=True`, a list of objects is returned.

If no edge exists, raise an exception.

`has_edge(self, n1, n2=None, x=None)`

Return True if digraph contains directed edge `(n1,n2,x)`.

Can be called as `G.has_edge(n1,n2,x)`
or as `G.has_edge(e)`, where `e=(n1,n2,x)`.

If `x` is unspecified, i.e. if called with an edge of the form `e=(n1,n2)`, then return True if there exists ANY edge from `n1` to `n2` (equivalent to `has_successor(n1,n2)`).

Overrides: `networkx.base.Graph.has_edge`

`has_neighbor(self, n1, n2)`

Return True if node `n1` and `n2` are connected.

True if there exists ANY edge `(n1,n2,x)` or `(n2,n1,x)` for some `x`.

Overrides: `networkx.base.Graph.has_neighbor`

`has_predecessor(self, n1, n2)`

Return True if node `n1` has a predecessor `n2`.

Return True if there exists ANY edge `(n2,n1,x)` for some `x`.

`has_successor(self, n1, n2)`

Return True if node `n1` has a successor `n2`.

Return True if there exists ANY edge `(n1,n2,x)` for some `x`.

`in_degree(self, nbunch=None, with_labels=False)`

Return the in-degree of single node or of `nbunch` of nodes.

If `nbunch` is omitted or `nbunch=None`, then return in-degrees of all nodes.

If `with_labels=True`, then return a dict that maps each `n` in `nbunch` to `in_degree(n)`.

Any nodes in `nbunch` that are not in the graph are (quietly) ignored.

Overrides: `networkx.base.DiGraph.in_degree`

`neighbors(self, n, with_labels=False)`

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

If `with_labels=True`, return a dict keyed by neighbors to edge data for that edge. If neighbor has both in and out edge, the edge data is returned as the list `[indata, outdata]`

The node `n` will be a neighbor of itself if a selfloop exists.

Overrides: `networkx.base.DiGraph.neighbors`

`neighbors_iter(self, n, with_labels=False)`

Return an iterator for neighbors of `n`.

If `with_labels=True`, the iterator returns `(neighbor, edge_data)` tuples for each edge. If neighbor has both in and out edges, the edge data is either:

- 1) concatenated as lists if `multiedges==True`, or
- 2) returned as the list `[indata, outdata]` if `multiedges==False`.

The node `n` will be a neighbor of itself if a selfloop exists.

Overrides: `networkx.base.DiGraph.neighbors_iter`

`nodes_with_selfloops(self)`

Return list of all nodes having self-loops.

`number_of_selfloops(self)`

Return number of self-loops in graph.

`out_degree(self, nbunch=None, with_labels=False)`

Return the out-degree of single node or of nbunch of nodes.
If nbunch is omitted or nbunch=None, then return
out-degrees of all nodes.

If with_labels=True, then return a dict that maps each n
in nbunch to out_degree(n).

Any nodes in nbunch that are not in the graph are
(quietly) ignored.

Overrides: `networkx.base.DiGraph.out_degree`

`predecessors(self, n, with_labels=False)`

Return a list of predecessor nodes of node n.

If with_labels=True, return a dict keyed by predecessors to
edge data for that edge.

Overrides: `networkx.base.DiGraph.predecessors`

`selfloop_edges(self)`

Return all edges that are self-loops.

subgraph(*self*, *nbunch*, *inplace=False*, *create_using=None*)

Return the subgraph induced on nodes in *nbunch*.

nbunch: either a singleton node, a string (which is treated as a singleton node), or any non-string iterable or iterator. For example, a list, dict, set, Graph, numeric array, or user-defined iterable object.

Setting *inplace=True* will return induced subgraph in original graph by deleting nodes not in *nbunch*. It overrides any setting of *create_using*.

WARNING: specifying *inplace=True* makes it easy to destroy the graph.

Unless otherwise specified, return a new graph of the same type as *self*. Use (optional) *create_using=R* to return the resulting subgraph in *R*. *R* can be an existing graph-like object (to be emptied) or *R* can be a call to a graph object, e.g. *create_using=DiGraph()*. See documentation for *empty_graph()*

Note: use *subgraph(G)* rather than *G.subgraph()* to access the more general *subgraph()* function from the operators module.

Overrides: `networkx.base.Graph.subgraph`

successors(*self*, *n*, *with_labels=False*)

Return a list of all successor nodes of node *n*.

If *with_labels=True*, return a dict keyed by successors to edge data for that edge.

Overrides: `networkx.base.DiGraph.successors`

to_undirected(self)

Return the underlying graph of G.

The underlying graph is its undirected representation: each directed edge is replaced with an undirected edge.

If multiedges=True, then an XDiGraph with only two directed edges (1,2,"red") and (2,1,"blue") will be converted into an XGraph with two undirected edges (1,2,"red") and (1,2,"blue"). Two directed edges (1,2,"red") and (2,1,"red") will result in in two undirected edges (1,2,"red") and (1,2,"red").

If multiedges=False, then two directed edges (1,2,"red") and (2,1,"blue") can only result in one undirected edge, and there is no guarantee which one it is.

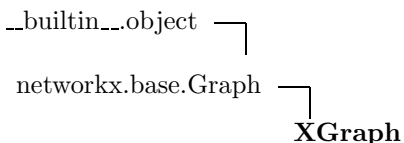
Overrides: networkx.base.DiGraph.to_undirected

Inherited from object: __delattr__, __getattr__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__

Inherited from DiGraph: __getitem__, add_node, add_nodes_from, clear, degree_iter, delete_node, delete_nodes_from, foo, in_degree_iter, is_directed, out_degree_iter, predecessors_iter, reverse, successors_iter, to_directed

Inherited from Graph: __contains__, __iter__, __len__, __str__, add_cycle, add_path, edge_boundary, edges, has_node, node_boundary, nodes, nodes_iter, number_of_edges, number_of_nodes, order, print_dna, size

31.3 Class XGraph



A class implementing general undirected graphs, allowing (optional) self-loops, multiple edges, arbitrary (hashable) objects as nodes and arbitrary objects associated with edges.

An XGraph edge is specified by a 3-tuple $e=(n1,n2,x)$, where $n1$ and $n2$ are (hashable) objects (nodes) and x is an arbitrary (and not necessarily unique) object associated with that edge.

```
>>> G=XGraph()
```

creates an empty simple and undirected graph (no self-loops or multiple edges allowed). It is equivalent to the expression:

```
>>> G=XGraph(name="No Name",selfloops=False,multiedges=False)
```

```
>>> G=XGraph(name="empty",multiedges=True)
```

creates an empty graph with `G.name="empty"`, that do not allow the addition of self-loops but do allow for multiple edges.

See also the `XDiGraph` class below.

31.3.1 Methods

<code>__init__(self, **kws)</code>
Initialize <code>XGraph</code> .
Optional arguments:: name: graph name (default="No Name") selfloops: if True selfloops are allowed (default=False) multiedges: if True multiple edges are allowed (default=False) Overrides: <code>networkx.base.Graph.__init__</code>
<code>add_cycle(self, nlist)</code>
Add the cycle of nodes in <code>nlist</code> to graph Overrides: <code>networkx.base.Graph.add_cycle</code>

`add_edge(self, n1, n2=None, x=None)`

Add a single edge to the graph.

Can be called as `G.add_edge(n1,n2,x)`
or as `G.add_edge(e)`, where `e=(n1,n2,x)`.

`n1,n2` are (hashable) node objects, and are added silently to the Graph if not already present.

`x` is an arbitrary (not necessarily hashable) object associated with this edge. It can be used to associate one or more: labels, data records, weights or any arbitrary objects to edges.

For example, if the graph `G` was created with

```
>>> G=XGraph()
```

then `G.add_edge(1,2,"blue")` will add the edge `(1,2,"blue")`.

If `G.multiedges=False`, then a subsequent `G.add_edge(1,2,"red")` will change the above edge `(1,2,"blue")` into the edge `(1,2,"red")`.

If `G.multiedges=True`, then two successive calls to `G.add_edge(1,2,"red")` will result in 2 edges of the form `(1,2,"red")` that can not be distinguished from one another.

`G.add_edge(1,2,"green")` will add both edges `(1,2,X)` and `(2,1,X)`.

If `self.selfloops=False`, then calling `add_edge(n1,n1,x)` will have no effect on the Graph.

Objects associated to an edge can be retrieved using `edges()`, `edge_iter()`, or `get_edge()`.

Overrides: `networkx.base.Graph.add_edge`

`add_edges_from(self, ebunch)`

Add multiple edges to the graph.

`ebunch`: Container of edges. Each edge must be a 3-tuple `(n1,n2,x)` or a 2-tuple `(n1,n2)`. See `add_edge` documentation.

The container must be iterable or an iterator. It is iterated over once.

Overrides: `networkx.base.Graph.add_edges_from`

`add_path(self, nlist)`

Add the path through the nodes in `nlist` to graph

Overrides: `networkx.base.Graph.add_path`

`allow_multiedges(self)`

Henceforth allow addition of multiedges (more than one edge between two nodes).

Warning: This causes all edge data to be converted to lists.

`allow_selfloops(self)`

Henceforth allow addition of self-loops (edges from a node to itself).

This doesn't change the graph structure, only what you can do to it.

`ban_multiedges(self)`

Remove multiedges retaining the data from the first edge.

Henceforth do not allow multiedges.

`ban_selfloops(self)`

Remove self-loops from the graph and henceforth do not allow their creation.

`copy(self)`

Return a (shallow) copy of the graph.

Return a new `XGraph` with same name and same attributes for selfloop and multiedges. Each node and each edge in original graph are added to the copy.

Overrides: `networkx.base.Graph.copy`

`degree(self, nbunch=None, with_labels=False)`

Return degree of single node or of nbunch of nodes.

If nbunch is omitted or nbunch=None, then return degrees of all nodes.

The degree of a node is the number of edges attached to that node.

Can be called in three ways:

`G.degree(n):` return the degree of node `n`

`G.degree(nbunch):` return a list of values,

one for each `n` in `nbunch`

(`nbunch` is any iterable container of nodes.)

`G.degree():` same as `nbunch = all nodes in graph`.

Always return a list.

If `with_labels==True`, then return a dict that maps each `n` in `nbunch` to `degree(n)`.

Any nodes in `nbunch` that are not in the graph are (quietly) ignored.

Overrides: `networkx.base.Graph.degree`

`degree_iter(self, nbunch=None, with_labels=False)`

This is the `degree()` method returned in iterator form.

If `with_labels=True`, iterator yields 2-tuples of form `(n,degree(n))` (like `iteritems()` on a dict.)

Overrides: `networkx.base.Graph.degree_iter`

`delete_edge(self, n1, n2=None, x=None)`

Delete the edge `(n1,n2,x)` from the graph.

Can be called either as `G.delete_edge(n1,n2,x)` or as `G.delete_edge(e)`, where `e=(n1,n2,x)`.

If `x` is unspecified, i.e. if called with an edge `e=(n1,n2)`, or as `G.delete_edge(n1,n2)`, then delete all edges between `n1` and `n2`.

If the edge does not exist, do nothing.

Overrides: `networkx.base.Graph.delete_edge`

`delete_edges_from(self, ebunch)`

Delete edges in ebunch from the graph.

ebunch: Container of edges. Each edge must be a 3-tuple (n1,n2,x) or a 2-tuple (n1,n2). In the latter case all edges between n1 and n2 will be deleted. See `delete_edge` above.

The container must be iterable or an iterator, and is iterated over once. Edges that are not in the graph are ignored.

Overrides: `networkx.base.Graph.delete_edges_from`

`delete_multiedges(self)`

Remove multiedges retaining the data from the first edge

`delete_node(self, n)`

Delete node n from graph.

Attempting to delete a non-existent node will raise an exception.

Overrides: `networkx.base.Graph.delete_node`

`delete_nodes_from(self, nbunch)`

Remove nodes in nbunch from graph.

nbunch: an iterable or iterator containing valid(hashable) node names.

Attempting to delete a non-existent node will raise an exception. This could result in a partial deletion of those nodes both in nbunch and in the graph.

Overrides: `networkx.base.Graph.delete_nodes_from`

`delete_selfloops(self)`

Remove self-loops from the graph (edges from a node to itself).

edges(*self*, *nbunch*=None, *with_labels*=False)

Return a list of all edges that originate at a node in *nbunch*, or a list of all edges if *nbunch*=None.

See `add_node` for definition of *nbunch*.

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

with_labels=True option is not supported because in that case you should probably use `neighbors()`.

Overrides: `networkx.base.Graph.edges`

edges_iter(*self*, *nbunch*=None, *with_labels*=False)

Return iterator that iterates once over each edge adjacent to nodes in *nbunch*, or over all nodes in graph if *nbunch*=None.

See `add_node` for definition of *nbunch*.

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

with_labels=True option is not supported (in that case you should probably use `neighbors()`).

Overrides: `networkx.base.Graph.edges_iter`

get_edge(*self*, *n1*, *n2*)

Return the objects associated with each edge between *n1* and *n2*.

If *multiedges*=False, a single object is returned.

If *multiedges*=True, a list of objects is returned.

If no edge exists, raise an exception.

has_edge(*self*, *n1*, *n2*=None, *x*=None)

Return True if graph contains edge (*n1*,*n2*,*x*).

Can be called as `G.has_edge(n1,n2,x)`

or as `G.has_edge(e)`, where `e=(n1,n2,x)`.

If *x* is unspecified, i.e. if called with an edge of the form `e=(n1,n2)`, then return True if there exists ANY edge between *n1* and *n2* (equivalent to `has_neighbor(n1,n2)`)

Overrides: `networkx.base.Graph.has_edge`

has_neighbor(*self*, *n1*, *n2*)

Return True if node *n1* has neighbor *n2*.

Note that this returns True if there exists ANY edge (*n1*,*n2*,*x*) for some *x*.

Overrides: `networkx.base.Graph.has_neighbor`

neighbors(*self*, *n*, *with_labels*=False)

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

If *with_labels*=True, return a dict keyed by neighbors to edge data for that edge.

Overrides: `networkx.base.Graph.neighbors`

neighbors_iter(*self*, *n*, *with_labels*=False)

Return an iterator for neighbors of *n*.

Overrides: `networkx.base.Graph.neighbors_iter`

nodes_with_selfloops(*self*)

Return list of all nodes having self-loops.

number_of_edges(*self*)

Return number of edges

Overrides: `networkx.base.Graph.number_of_edges`

number_of_selfloops(*self*)

Return number of self-loops in graph.

selfloop_edges(*self*)

Return all edges that are self-loops.

size(*self*)

Return the size of a graph = number of edges.

Overrides: `networkx.base.Graph.size`

subgraph(*self*, *nbunch*, *inplace=False*, *create_using=None*)

Return the subgraph induced on nodes in *nbunch*.

nbunch: either a singleton node, a string (which is treated as a singleton node), or any non-string iterable or iterator. For example, a list, dict, set, Graph, numeric array, or user-defined iterable object.

Setting *inplace=True* will return induced subgraph in original graph by deleting nodes not in *nbunch*. It overrides any setting of *create_using*.

WARNING: specifying *inplace=True* makes it easy to destroy the graph.

Unless otherwise specified, return a new graph of the same type as *self*. Use (optional) *create_using=R* to return the resulting subgraph in *R*. *R* can be an existing graph-like object (to be emptied) or *R* can be a call to a graph object, e.g. *create_using=DiGraph()*. See documentation for *empty_graph()*

Note: use *subgraph(G)* rather than *G.subgraph()* to access the more general *subgraph()* function from the operators module.

Overrides: *networkx.base.Graph.subgraph*

to_directed(*self*)

Return a directed representation of the XGraph *G*.

A new *XDigraph* is returned with the same name, same nodes and with each edge (*u,v,x*) replaced by two directed edges (*u,v,x*) and (*v,u,x*).

Overrides: *networkx.base.Graph.to_directed*

Inherited from object: *__delattr__*, *__getattr__*, *__hash__*, *__new__*, *__reduce__*, *__reduce_ex__*, *__repr__*, *__setattr__*

Inherited from Graph: *__contains__*, *__getitem__*, *__iter__*, *__len__*, *__str__*, *add_node*, *add_nodes_from*, *clear*, *edge_boundary*, *has_node*, *is_directed*, *node_boundary*, *nodes*, *nodes_iter*, *number_of_nodes*, *order*, *print_dna*, *to_undirected*

Index

- networkx (*package*), 2–4
- networkx.base (*module*), 5–26
 - degree (*function*), 9
 - degree_histogram (*function*), 9
 - density (*function*), 9
 - DiGraph (*class*), 10–18
 - __getitem__ (*method*), 12
 - __init__ (*method*), 12
 - add_edge (*method*), 12
 - add_edges_from (*method*), 12
 - add_node (*method*), 13
 - add_nodes_from (*method*), 13
 - clear (*method*), 13
 - copy (*method*), 13
 - degree (*method*), 14
 - degree_iter (*method*), 14
 - delete_edge (*method*), 14
 - delete_edges_from (*method*), 15
 - delete_node (*method*), 15
 - delete_nodes_from (*method*), 15
 - edges_iter (*method*), 15
 - foo (*method*), 16
 - in_degree (*method*), 16
 - in_degree_iter (*method*), 16
 - is_directed (*method*), 16
 - neighbors (*method*), 16
 - neighbors_iter (*method*), 16
 - out_degree (*method*), 17
 - out_degree_iter (*method*), 17
 - predecessors (*method*), 17
 - predecessors_iter (*method*), 17
 - reverse (*method*), 17
 - successors (*method*), 17
 - successors_iter (*method*), 17
 - to_directed (*method*), 17
 - to_undirected (*method*), 17
 - edges (*function*), 9
 - edges_iter (*function*), 9
- Graph (*class*), 18–26
 - __contains__ (*method*), 18
 - __getitem__ (*method*), 18
 - __init__ (*method*), 18
 - __iter__ (*method*), 19
 - __len__ (*method*), 19
 - __str__ (*method*), 19
 - add_cycle (*method*), 19
 - add_edge (*method*), 19
 - add_edges_from (*method*), 19
 - add_node (*method*), 20
 - add_nodes_from (*method*), 20
 - add_path (*method*), 20
 - clear (*method*), 21
 - copy (*method*), 21
 - degree (*method*), 21
 - degree_iter (*method*), 21
 - delete_edge (*method*), 22
 - delete_edges_from (*method*), 22
 - delete_node (*method*), 22
 - delete_nodes_from (*method*), 22
 - edge_boundary (*method*), 23
 - edges (*method*), 23
 - edges_iter (*method*), 23
 - has_edge (*method*), 23
 - has_neighbor (*method*), 24
 - has_node (*method*), 24
 - is_directed (*method*), 24
 - neighbors (*method*), 24
 - neighbors_iter (*method*), 24
 - node_boundary (*method*), 24
 - nodes (*method*), 24
 - nodes_iter (*method*), 24
 - number_of_edges (*method*), 25
 - number_of_nodes (*method*), 25
 - order (*method*), 25
 - print_dna (*method*), 25
 - size (*method*), 25
 - subgraph (*method*), 25
 - to_directed (*method*), 25
 - to_undirected (*method*), 26
- neighbors (*function*), 9
- NetworkXError (*class*), 26
- NetworkXException (*class*), 26
- nodes (*function*), 10
- nodes_iter (*function*), 10
- number_of_edges (*function*), 10
- number_of_nodes (*function*), 10
- networkx centrality (*module*), 27–28
 - betweenness centrality (*function*), 27
 - closeness centrality (*function*), 27
 - degree centrality (*function*), 27
 - edge_betweenness (*function*), 27
- networkx cliques (*module*), 29–32
 - cliques_containing_node (*function*), 29
 - find_cliques (*function*), 29

- graph_clique_number (*function*), 30
- graph_number_of_cliques (*function*), 30
- make_clique_bipartite (*function*), 30
- make_max_clique_graph (*function*), 31
- node_clique_number (*function*), 31
- number_of_cliques (*function*), 31
- project_down (*function*), 31
- project_up (*function*), 32
- networkx.cluster (*module*), 33–34
 - average_clustering (*function*), 33
 - clustering (*function*), 33
 - transitivity (*function*), 33
 - triangles (*function*), 33
- networkx.cores (*module*), 35
 - find_cores (*function*), 35
- networkx.drawing (*package*), 36
- networkx.drawing.layout (*module*), 37–38
 - circular_layout (*function*), 37
 - graph_low_ev_pi (*function*), 37
 - random_layout (*function*), 37
 - shell_layout (*function*), 37
 - spectral_layout (*function*), 37
 - spring_layout (*function*), 37
- networkx.drawing.nx_pydot (*module*), 39–40
 - networkx_from_pydot (*function*), 39
 - pydot_from_networkx (*function*), 39
 - pydot_layout (*function*), 39
 - read_dot (*function*), 39
 - write_dot (*function*), 39
- networkx.drawing.nx_pydot (*module*), 41–43
 - draw (*function*), 41
 - draw_circular (*function*), 41
 - draw_nx (*function*), 41
 - draw_nxpydot (*function*), 41
 - draw_nxpydot_nolabels (*function*), 41
 - draw_pydot (*function*), 42
 - draw_pydot_subgraph (*function*), 42
 - draw_random (*function*), 42
 - draw_shell (*function*), 42
 - draw_spectral (*function*), 42
 - draw_spring (*function*), 42
 - drawg (*function*), 42
 - mpl_network (*function*), 42
- networkx.drawing.nx_vtk (*module*), 44
 - draw_nxvtk (*function*), 44
- networkx.generators (*package*), 45
- networkx.generators.atlas (*module*), 46
 - graph_atlas_g (*function*), 46
- networkx.generators.classic (*module*), 47–51
 - balanced_tree (*function*), 47
 - barbell_graph (*function*), 47
 - circular_ladder_graph (*function*), 47
 - complete_bipartite_graph (*function*), 48
 - complete_graph (*function*), 48
 - cycle_graph (*function*), 48
 - dorogovtsev_goltsev_mendes_graph (*function*), 48
 - empty_graph (*function*), 48
 - grid_2d_graph (*function*), 49
 - grid_graph (*function*), 49
 - hypercube_graph (*function*), 50
 - ladder_graph (*function*), 50
 - lollipop_graph (*function*), 50
 - null_graph (*function*), 50
 - path_graph (*function*), 50
 - periodic_grid_2d_graph (*function*), 50
 - star_graph (*function*), 51
 - trivial_graph (*function*), 51
 - wheel_graph (*function*), 51
- networkx.generators.degree_seq (*module*), 52–55
 - configuration_model (*function*), 52
 - create_degree_sequence (*function*), 52
 - degree_sequence_tree (*function*), 53
 - havel_hakimi_graph (*function*), 53
 - is_valid_degree_sequence (*function*), 54
- networkx.generators.geometric (*module*), 56
 - random_geometric_graph (*function*), 56
- networkx.generators.random_graphs (*module*), 57–62
 - barabasi_albert_graph (*function*), 57
 - binomial_graph (*function*), 57
 - erdos_renyi_graph (*function*), 57
 - newman_watts_strogatz_graph (*function*), 58
 - powerlaw_cluster_graph (*function*), 58
 - random_lobster (*function*), 59
 - random_powerlaw_tree (*function*), 59
 - random_powerlaw_tree_sequence (*function*), 60
 - random_regular_graph (*function*), 60
 - random_shell_graph (*function*), 61
 - watts_strogatz_graph (*function*), 62
- networkx.generators.small (*module*), 63–67
 - bull_graph (*function*), 63
 - chvatal_graph (*function*), 63
 - cubical_graph (*function*), 63
 - desargues_graph (*function*), 63
 - diamond_graph (*function*), 63
 - dodecahedral_graph (*function*), 63
 - frucht_graph (*function*), 63

- heawood_graph (*function*), 63
- house_graph (*function*), 63
- house_x_graph (*function*), 63
- icosahedral_graph (*function*), 64
- krackhardt_kite_graph (*function*), 64
- LCF_graph (*function*), 64
- make_small_graph (*function*), 65
- moebius_kantor_graph (*function*), 66
- octahedral_graph (*function*), 66
- pappus_graph (*function*), 66
- petersen_graph (*function*), 66
- sedgewick_maze_graph (*function*), 66
- tetrahedral_graph (*function*), 67
- truncated_cube_graph (*function*), 67
- truncated_tetrahedron_graph (*function*), 67
- tutte_graph (*function*), 67
- networkx.hybrid (*module*), 68
 - is_kl_connected (*function*), 68
 - kl_connected_subgraph (*function*), 68
- networkx.io (*module*), 69–71
 - read_adjlist (*function*), 69
 - read_edgelist (*function*), 69
 - read_gpickle (*function*), 69
 - read_multiline_adjlist (*function*), 69
 - write_adjlist (*function*), 70
 - write_edgelist (*function*), 70
 - write_gpickle (*function*), 70
 - write_multiline_adjlist (*function*), 70
- networkx.isomorph (*module*), 72
 - fast_graph_could_be_isomorphic (*function*), 72
 - faster_graph_could_be_isomorphic (*function*), 72
 - graph_could_be_isomorphic (*function*), 72
- networkx.operators (*module*), 73–76
 - cartesian_product (*function*), 73
 - complement (*function*), 73
 - compose (*function*), 73
 - convert_node_labels_to_integers (*function*), 73
 - convert_to_directed (*function*), 74
 - convert_to_undirected (*function*), 74
 - create_empty_copy (*function*), 74
 - disjoint_union (*function*), 74
 - subgraph (*function*), 75
 - union (*function*), 75
- networkx.paths (*module*), 77–80
 - center (*function*), 77
 - diameter (*function*), 77
 - dijkstra (*function*), 77
 - dijkstra_path (*function*), 77
 - dijkstra_path_length (*function*), 78
 - eccentricity (*function*), 78
 - is_directed_acyclic_graph (*function*), 78
 - periphery (*function*), 78
 - radius (*function*), 78
 - shortest_path (*function*), 79
 - shortest_path_length (*function*), 79
 - topological_sort (*function*), 79
 - topological_sort_recursive (*function*), 79
- networkx.queues (*module*), 81–84
 - BFS (*class*), 81–82
 - __init__ (*method*), 81
 - update (*method*), 81
 - DFS (*class*), 82
 - __init__ (*method*), 82
 - update (*method*), 82
 - FIFO (*class*), 82–83
 - __init__ (*method*), 82
 - pop (*method*), 82
 - LIFO (*class*), 83
 - __init__ (*method*), 83
 - Priority (*class*), 83
 - __init__ (*method*), 83
 - __len__ (*method*), 83
 - append (*method*), 83
 - extend (*method*), 83
 - pop (*method*), 83
 - smallest (*method*), 83
 - Random (*class*), 83–84
 - __init__ (*method*), 84
 - pop (*method*), 84
 - RFS (*class*), 84
 - __init__ (*method*), 84
 - update (*method*), 84
- networkx.release (*module*), 85
- networkx.search (*module*), 86–88
 - bfs_length (*function*), 86
 - bfs_path (*function*), 86
 - connected_component_subgraphs (*function*), 87
 - connected_components (*function*), 87
 - dfs_forest (*function*), 87
 - dfs_postorder (*function*), 87
 - dfs_predecessor (*function*), 87
 - dfs_preorder (*function*), 87
 - dfs_successor (*function*), 87
 - is_connected (*function*), 87
 - node_connected_component (*function*), 88
 - number_connected_components (*function*), 88

- networkx.search_class (*module*), 89–96
 - Forest (*class*), 89–90
 - __init__ (*method*), 90
 - end_tree (*method*), 90
 - lastseen_edge (*method*), 90
 - start_tree (*method*), 90
 - Length (*class*), 90–91
 - __init__ (*method*), 90
 - lastseen_edge (*method*), 90
 - Postorder (*class*), 91
 - __init__ (*method*), 91
 - end_tree (*method*), 91
 - lastseen_vertex (*method*), 91
 - start_tree (*method*), 91
 - Predecessor (*class*), 91–92
 - __init__ (*method*), 92
 - firstseen_vertex (*method*), 92
 - lastseen_edge (*method*), 92
 - path (*method*), 92
 - Preorder (*class*), 92–93
 - __init__ (*method*), 93
 - end_tree (*method*), 93
 - firstseen_vertex (*method*), 93
 - start_tree (*method*), 93
 - Search (*class*), 93–95
 - __init__ (*method*), 94
 - end_tree (*method*), 94
 - firstseen_edge (*method*), 94
 - firstseen_vertex (*method*), 94
 - lastseen_edge (*method*), 94
 - lastseen_vertex (*method*), 94
 - search (*method*), 94
 - start_tree (*method*), 95
 - Successor (*class*), 95–96
 - __init__ (*method*), 95
 - firstseen_vertex (*method*), 95
 - lastseen_edge (*method*), 95
- networkx.spectrum (*module*), 97
 - adj_matrix (*function*), 97
 - generalized_laplacian (*function*), 97
 - laplacian (*function*), 97
- networkx.threshold (*module*), 98–104
 - betweenness_sequence (*function*), 98
 - cluster_sequence (*function*), 98
 - creation_sequence (*function*), 98
 - creation_sequence_to_weights (*function*), 98
 - degree_correlation (*function*), 99
 - degree_sequence (*function*), 99
 - density (*function*), 99
 - eigenvalues (*function*), 99
 - find_alternating_4_cycle (*function*), 99
 - find_creation_sequence (*function*), 99
 - find_threshold_graph (*function*), 100
 - is_threshold_graph (*function*), 100
 - is_threshold_sequence (*function*), 100
 - left_d_threshold_sequence (*function*), 100
 - make_compact (*function*), 100
 - random_threshold_sequence (*function*), 100
 - right_d_threshold_sequence (*function*), 101
 - shortest_path (*function*), 101
 - shortest_path_length (*function*), 101
 - swap_d (*function*), 102
 - threshold_graph (*function*), 102
 - triangle_sequence (*function*), 102
 - triangles (*function*), 102
 - uncompact (*function*), 103
 - weights_to_creation_sequence (*function*), 103
- networkx.utils (*module*), 105–106
 - discrete_sequence (*function*), 105
 - gsl_pareto_sequence (*function*), 105
 - gsl_poisson_sequence (*function*), 105
 - gsl_powerlaw_sequence (*function*), 105
 - gsl_uniform_sequence (*function*), 105
 - is_list_of_ints (*function*), 105
 - is_singleton (*function*), 105
 - is_string_like (*function*), 105
 - iterable (*function*), 105
 - pareto_sequence (*function*), 105
 - powerlaw_sequence (*function*), 105
 - scipy_discrete_sequence (*function*), 106
 - scipy_pareto_sequence (*function*), 106
 - scipy_poisson_sequence (*function*), 106
 - scipy_powerlaw_sequence (*function*), 106
 - scipy_uniform_sequence (*function*), 106
 - uniform_sequence (*function*), 106
- networkx.xbase (*module*), 107–130
 - XDiGraph (*class*), 112–122
 - __init__ (*method*), 114
 - add_edge (*method*), 114
 - add_edges_from (*method*), 115
 - allow_multiedges (*method*), 116
 - allow_selfloops (*method*), 116
 - ban_multiedges (*method*), 116
 - ban_selfloops (*method*), 116
 - copy (*method*), 116
 - degree (*method*), 116

- delete_edge (*method*), 117
- delete_edges_from (*method*), 117
- delete_multiedges (*method*), 117
- delete_selfloops (*method*), 117
- edges_iter (*method*), 117
- get_edge (*method*), 118
- has_edge (*method*), 118
- has_neighbor (*method*), 118
- has_predecessor (*method*), 118
- has_successor (*method*), 118
- in_degree (*method*), 119
- neighbors (*method*), 119
- neighbors_iter (*method*), 119
- nodes_with_selfloops (*method*), 119
- number_of_selfloops (*method*), 119
- out_degree (*method*), 120
- predecessors (*method*), 120
- selfloop_edges (*method*), 120
- subgraph (*method*), 120
- successors (*method*), 121
- to_undirected (*method*), 121
- XGraph (*class*), 122–130
 - __init__ (*method*), 123
 - add_cycle (*method*), 123
 - add_edge (*method*), 123
 - add_edges_from (*method*), 124
 - add_path (*method*), 124
 - allow_multiedges (*method*), 125
 - allow_selfloops (*method*), 125
 - ban_multiedges (*method*), 125
 - ban_selfloops (*method*), 125
 - copy (*method*), 125
 - degree (*method*), 125
 - degree_iter (*method*), 126
 - delete_edge (*method*), 126
 - delete_edges_from (*method*), 126
 - delete_multiedges (*method*), 127
 - delete_node (*method*), 127
 - delete_nodes_from (*method*), 127
 - delete_selfloops (*method*), 127
 - edges (*method*), 127
 - edges_iter (*method*), 128
 - get_edge (*method*), 128
 - has_edge (*method*), 128
 - has_neighbor (*method*), 128
 - neighbors (*method*), 129
 - neighbors_iter (*method*), 129
 - nodes_with_selfloops (*method*), 129
 - number_of_edges (*method*), 129
 - number_of_selfloops (*method*), 129
 - selfloop_edges (*method*), 129
 - size (*method*), 129
 - subgraph (*method*), 129
 - to_directed (*method*), 130