NX

API Documentation

April 10, 2005

Contents

1 Package NX

```
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 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 NX.base for the details of the API.
```

1.1 Modules

- **base**:
  `Base classes for graphs and digraphs.`
  (Section 2, p. 9)
- **centrality**:
  `Centrality measures.`
  (Section 3, p. 31)
- **cliques**:
  `Cliques - Find and manipulate cliques of graphs`
  (Section 4, p. 33)
- **cluster**:
  `Compute clustering coefficients and transitivity of graphs.`
  (Section 5, p. 37)
- **cores**:
  `Find and manipulate the k-cores of a graph`
  (Section 6, p. 39)
- **drawing** *(Section 7, p. 40)*
  - **layout**:
    `Layout (positioning) algorithms for graph drawing.`
    (Section 8, p. 41)
  - **nx_pydot**:
    `Import and export NX networks to dot format using pydot.`
    (Section 9, p. 43)
- **generators**:
  `A package for generating various graphs in NX.`

(Section 26, p. 93)
- **tests** *(Section 27, p. 94)*
  - **test** *(Section 28, p. 95)*
  - **test2** *(Section 29, p. 96)*
- **utils**:
  `Utilities for NX package`
  (Section 30, p. 97)
- **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. 100)

1.2 Variables

Name	Description
__author__	**Value:** `'Aric Hagberg <hagberg@lanl.gov>\nDan Schult <-dschult@colgate.edu>\nPieter Sw...` *(type=`str`)*
__date__	**Value:** `'Sun Apr 10 10:55:09 2005'` *(type=`str`)*
__license__	**Value:** `'LGPL'` *(type=`str`)*
__version__	**Value:** `'0.1.1-7'` *(type=`str`)*

2 Module NX.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,
mutiple 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,... to indicate a node,
v=vertex=node).

G,G1,G2,H,etc:

Graphs

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

nodes (v stands for vertex=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 NX 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(*G*, *nbunch*=None, *with_labels*=False)

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

degree_histogram(*G*)

```
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(*G*)

```
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(*G*, *nbunch*=None, *with_labels*=False)

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

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

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

neighbors(*G*, *n*, *with_labels*=`False`)

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

nodes(*G*)

`Return a copy of the graph nodes in a list.`

nodes_iter(*G*)

`Return an iterator over the graph nodes.`

number_of_edges(*G*)

`Return the size of a graph = number of edges.`

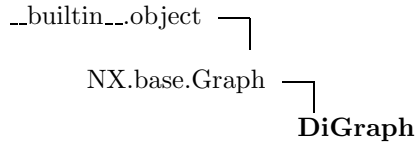
number_of_nodes(*G*)

`Return the order of a graph = number of nodes.`

2.2 Variables

Name	Description
__author__	**Value:** `'Aric Hagberg (hagberg@lanl.gov)\nPieter Swart - (swart@lanl.gov)\nDan Schult(d...` *(type=**str**)*
__credits__	**Value:** `''` *(type=**str**)*
__date__	**Value:** `'$Date: 2005/04/01 19:35:22 $'` *(type=**str**)*
__revision__	**Value:** `'$Revision: 1.104 $'` *(type=**str**)*

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*, ***kwds*)

Initialize Graph.

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

Overrides: NX.base.Graph.__init__ extit(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: NX.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 nor 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: NX.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: NX.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: NX.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: NX.base.Graph.add_nodes_from

clear(*self*)

```
Remove name and delete all nodes and edges from digraph.
```

Overrides: NX.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: NX.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: NX.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 ommitted, 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: NX.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: NX.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: NX.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: NX.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: NX.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: NX.base.Graph.edges_iter

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: NX.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: NX.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: NX.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 ommitted, 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.
```

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: NX.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: NX.base.Graph.to_undirected

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

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

2.3.2 Class Variables

Name	Description
Inherited from Graph: __slotnames__ *(p. 22)*	

2.4 Class Graph

__builtin__.object
└
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*, ***kwds*)

```
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 nor 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', <NX.baseNX.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*)

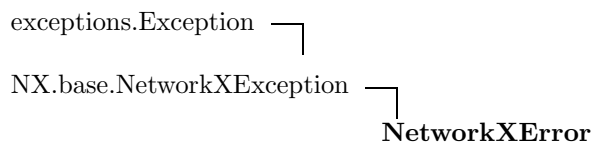
```
Return the undirected representation of the graph G.

This graph is undirected, so merely return a copy.
```

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

2.4.2 Class Variables

Name	Description
__slotnames__	**Value:** `[]` *(type=`list`)*

2.5 Class NetworkXError

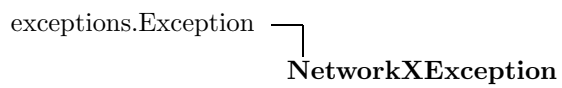


`Exception for a serious error in NetworkX`

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 NX.centrality

Centrality measures.

3.1 Functions

betweenness_centrality(*G*, *v*=False, *cutoff*=False)

```
Betweenness centrality for nodes.

>>> b=betweeness(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.

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.

Returns a dictionary of closeness centrality values keyed by node.
The closeness centrality is normalized to be bewtween [0,1].
```

degree_centrality(*G*, *v*=False)

```
Degree centrality for nodes.

Returns a dictionary of degree centrality values keyed by node.

The degree centrality is normalized to be bewtween [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
__author__	**Value:** `'Aric Hagberg (hagberg@lanl.gov)\nPieter Swart - (swart@lanl.gov)'` *(type=str)*
__credits__	**Value:** `''` *(type=str)*
__date__	**Value:** `'$Date: 2005/03/30 23:56:28 $'` *(type=str)*
__revision__	**Value:** `'$Revision: 1.9 $'` *(type=str)*

4 Module NX.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

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

```
Returns a list of cliques containing the given node.

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

find_cliques(*G*)

```
Find_cliques algorithm based on Bron & Kerbosch

This algorithm searchs 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, ***kwds*)

```
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, ***kwds*)

```
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, ***kwds*)

```
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, ***kwds*)

```
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
__author__	**Value:** 'Dan Schult (dschult@colgate.edu)' *(type=str)*
__credits__	**Value:** '' *(type=str)*
__date__	**Value:** '$Date: 2005/03/30 23:56:28 $' *(type=str)*
__revision__	**Value:** '$Revision: 1.27 $' *(type=str)*

5 Module NX.cluster

```
Compute clustering coefficients and transitivity of graphs.

Clustering coefficient

For each node find the fraction of possible triangles that are triangles,
c_i = 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 = 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*triangles/triads
```

5.1 Functions

average_clustering(G)

```
Average clustering coefficient for a graph.

Note: this is a space saving routine; It might be faster
to use clustering to get a list and then take average.
```

clustering(G, *nbunch*=None, ***kwds*)

```
Clustering coefficient for each node in nbunch
```

transitivity(G)

```
Transitivity (fraction of transitive triangles) for a graph
```

triangles(G, *nbunch*=None, ***kwds*)

```
Return number of triangles for nbunch of nodes.
If nbunch is None, then return triangles for every node.

Note: Each triangle is counted three times: once at each vertex.
```

5.2 Variables

Name	Description
__author__	**Value:** `'Aric Hagberg (hagberg@lanl.gov)\nPieter Swart - (swart@lanl.gov)\nDan Schult (...` *(type=`str`)*
__credits__	**Value:** `''` *(type=`str`)*
__date__	**Value:** `'$Date: 2005/03/30 23:56:28 $'` *(type=`str`)*
__revision__	**Value:** `'$Revision: 1.28 $'` *(type=`str`)*

6 Module NX.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
__author__	**Value:** 'Dan Schult(dschult@colgate.edu)' *(type=str)*
__credits__	**Value:** '' *(type=str)*
__date__	**Value:** '$Date: 2005/03/30 23:56:28 $' *(type=str)*
__revision__	**Value:** '$Revision: 1.15 $' *(type=str)*

7 Package NX.drawing

7.1 Modules

- **layout**:
  `Layout (positioning) algorithms for graph drawing.`
  (Section 8, p. 41)
- **nx_pydot**:
  `Import and export NX networks to dot format using pydot.`
  (Section 9, p. 43)

8 Module NX.drawing.layout

Layout (positioning) algorithms for graph drawing.

8.1 Functions

circular_layout(*G*)

Circular layout.

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

gershgorin_setup(*G*)

Return a list of matrix properties to be used
to iteratively multiply B*v where v is a vector
and B=g*I-L and g is the Gershgorin estimate of
the largest eigenvalue of L=Laplacian(G).

Used as input to graph_gershgorin_dot_v()

graph_gershgorin_dot_v(*gg_data*, *v*)

Returns B*v where B=g*I-L and g is the Gershgorin estimate of the
largest eigenvalue of L. (g=max(deg(n) + sum_u(|w_(n,u)|)

We use this to iterate and find the smallest eigenvectors of L.

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*)

Random layout.

shell_layout(*G*, *nlist*=None)

```
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*, *vpos*=None, *iterations*=1000, *eps*=0.001)

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

spring_layout(*G*, *iterations*=50, *vpos*=False)

```
Spring force model layout
```

8.2 Variables

Name	Description
__author__	**Value:** 'Aric Hagberg (hagberg@lanl.gov)\nDan Schult(dschult@colgate.edu)' *(type=**str**)*
__credits__	**Value:** '' *(type=**str**)*
__date__	**Value:** '$Date: 2005/03/30 23:56:28 $' *(type=**str**)*
__revision__	**Value:** '$Revision: 1.18 $' *(type=**str**)*

9 Module NX.drawing.nx_pydot

```
Import and export NX 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

NX_from_pydot(*D*, *result*=False)

```
Creates an NX graph from an pydot graph D
```

pydot_from_NX(*N*)

```
Creates a pydot graph from an NX graph N
```

pydot_layout(*G*, ***kwds*)

```
Create layout using pydot and graphviz.
Returns a dictionary of positions keyed by node.

>>> pos=pydot_layout(G)
>>> pos=pydot_layout(G,prog="twopi")
```

read_dot(*path*=False)

```
Creates an NX graph from a dot file
```

write_dot(*G*, *path*=False)

```
Write G to a graphviz dot file.
```

9.2 Variables

Name	Description
__author__	**Value:** 'Aric Hagberg (hagberg@lanl.gov)' *(type=**str**)*
__credits__	**Value:** '' *(type=**str**)*
__date__	**Value:** '$Date: 2005/04/01 18:36:44 $' *(type=**str**)*
__revision__	**Value:** '$Revision: 1.6 $' *(type=**str**)*

10 Package NX.generators

A package for generating various graphs in NX.

10.1 Modules

- **atlas**:
  Generators for the small graph atlas.
  (Section 11, p. 45)
- **classic**:
  Generators for some classic graphs.
  (Section 12, p. 46)
- **degree_seq**:
  Generate graphs with a given degree sequence.
  (Section 13, p. 51)
- **geometric**:
  Generators for geometric graphs.
  (Section 14, p. 55)
- **random_graphs**:
  Generators for random graphs
  (Section 15, p. 56)
- **small**:
  Various small and named graphs, together with some compact generators.
  (Section 16, p. 61)

11 Module NX.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.
```

11.1 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)
```

11.2 Variables

Name	Description
__author__	**Value:** 'Pieter Swart (swart@lanl.gov)' *(type=str)*
__credits__	**Value:** '' *(type=str)*
__date__	**Value:** '$Date: 2005/03/30 23:56:28 $' *(type=str)*
__revision__	**Value:** '$Revision: 1.14 $' *(type=str)*

12 Module NX.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,..,100
as a simple graph. Except for empty_graph, all these generators return
a Graph class (i.e. a simple undirected graph).
```

12.1 Functions

balanced_tree(*r*, *h*)

```
Return the perfectly balanced r-tree of height h.

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

Graph order=1+r+r**2+...+r**h=(r**(h+1)-1)/(r-1), graph size=order-1.

Node labels are integers numbered 1 (the root) up to order.
```

barbell_graph(*m1*, *m2*)

```
Return the Barbell Graph: two complete graphs connected by a path.

For m1>1 and m2>=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,...,m1 for the left barbell,
m1+1,...,m1+m2 for the path, and m1+m2+1,...,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.
```

empty_graph(*n*=0, *create_using*=None, ***kwds*)

```
Return the empty graph with n nodes
(with integer labels 1,...,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 mxn 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>=0, the complete graph K_m is connected to the
path P_n.  The resulting m+n nodes are labelled 1,...,m for the
complete graph and m+1,...,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, ***kwds*)

```
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*, *create_using*=None, ***kwds*)

```
Return the 2-D Grid Graph of mxn 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.
```

12.2 Variables

Name	Description
__author__	**Value:** 'Aric Hagberg (hagberg@lanl.gov)\nPieter Swart - (swart@lanl.gov)' *(type=str)*
__credits__	**Value:** '' *(type=str)*
__date__	**Value:** '$Date: 2005/04/08 19:02:59 $' *(type=str)*
__revision__	**Value:** '$Revision: 1.36 $' *(type=str)*

13 Module NX.generators.degree_seq

Generate graphs with a given degree sequence.

13.1 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, ***kwds*)

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

n: length of degree sequence = number of nodes

sfunction: a function, called as "sfunction(n,kwds)",

that returns a list of n real or integer values.

max_tries: max number of attempts at creating valid degree

sequence.

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

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

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

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.
```

13.2 Variables

Name	Description
__author__	**Value:** 'Aric Hagberg (hagberg@lanl.gov)\nPieter Swart - (swart@lanl.gov)\nDan Schult (... *(type=str)*
__credits__	**Value:** '' *(type=str)*
__date__	**Value:** '$Date: 2005/03/30 23:56:28 $' *(type=str)*
__revision__	**Value:** '$Revision: 1.42 $' *(type=str)*

14 Module NX.generators.geometric

```
Generators for geometric graphs.
```

14.1 Functions

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

```
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.
```

14.2 Variables

Name	Description
__author__	**Value:** `'Aric Hagberg (hagberg@lanl.gov)\nPieter Swart - (swart@lanl.gov)'` *(type=`str`)*
__credits__	**Value:** `''` *(type=`str`)*
__date__	**Value:** `'$Date: 2005/03/30 23:56:28 $'` *(type=`str`)*
__revision__	**Value:** `'$Revision: 1.14 $'` *(type=`str`)*

15 Module NX.generators.random_graphs

Generators for random graphs

15.1 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.

Note: The initialization of the B-A model is not specified
and we use a single node. This node is special in that it
gets a chance of attachment prop to k+1 instead of k.

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)
```

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_regular_graph(*d*, *n*, *seed*=None)

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

n*d must be even

Nodes are numbered 0...n-1.
To get a uniform sample from the space of random graphs
you should chose d<n^{1/3}.
.
For algorith see Kim and Vu's paper.

Reference:

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

The algorithm is based on an earlier paper:

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

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.
```

Parameters

```
    n:    the number of nodes
    k:    each vertex is connected to k neighbors in the circular topology
    p:    the probability of rewiring
    seed: seed for random number generator (default=None)
```

15.2 Variables

Name	Description
__author__	**Value:** 'Aric Hagberg (hagberg@lanl.gov)\nPieter Swart - (swart@lanl.gov)\nDan Schult(d... *(type=str)*
__credits__	**Value:** '' *(type=str)*
__date__	**Value:** '$Date: 2005/03/30 23:56:28 $' *(type=str)*
__revision__	**Value:** '$Revision: 1.17 $' *(type=str)*

16 Module NX.generators.small

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

16.1 Functions

bull_graph()

Return the Bull graph.

chvatal_graph()

Return the Chvatal Graph.

cubical_graph()

Return the 3-regular Platonic Cubical Graph.

desargues_graph()

Return the Desargues graph.

diamond_graph()

Return the Diamond Graph.

dodecahedral_graph()

Return the Platonic Dodecahedral graph.

frucht_graph()

Return the Frucht Graph.

The Frucht Graph is the smallest cubical graph whose
automorphism group consists only of the identity element.

heawood_graph()

Return the Heawood graph, a (3,6) cage.

house_graph()

Return the House graph (square with triangle on top).

house_x_graph()

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

icosahedral_graph()

```
Return the Platonic Icosahedral Graph.
```

krackhardt_kite_graph()

```
Return the Krackhardt Kite Social Network.

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 = [s1,s2,..,sk], a list of integer shifts mod n,

repeats

integer specifying the number of times that shifts in shift_list
are successively applied to each v_current in the n-cycle
to generate an edge between v_current and v_current+shift mod n.

For v1 cycling through the n-cycle a total of k*repeats
with shift cycling through shiftlist repeats times connect
v1 with v1+shift mod n

The utility graph K_{3,3}

>>> G=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, ***kwds*)

```
Return the small graph described by graph_description and kwds.

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 C_4 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 C_4 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 kwds
```

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 Tetrahedral Graph.
```

tutte_graph()

```
Return the Tutte Graph.
```

16.2 Variables

Name	Description
__author__	**Value:** 'Aric Hagberg (hagberg@lanl.gov)\nPieter Swart - (swart@lanl.gov)' *(type=str)*
__credits__	**Value:** '' *(type=str)*
__date__	**Value:** '$Date: 2005/04/10 04:52:35 $' *(type=str)*
__revision__	**Value:** '$Revision: 1.24 $' *(type=str)*

17 Module NX.hybrid

Hybrid

17.1 Functions

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

Returns True if G is kl connected

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

```
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
```

17.2 Variables

Name	Description
__author__	**Value:** 'Aric Hagberg (hagberg@lanl.gov)\nDan Schult (d-schult@colgate.edu)' *(type=str)*
__credits__	**Value:** '' *(type=str)*
__date__	**Value:** '$Date: 2005/03/30 23:56:28 $' *(type=str)*
__revision__	**Value:** '$Revision: 1.12 $' *(type=str)*

18 Module NX.io

Read and write graphs and networks.

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

18.1 Functions

read_adjlist(*path*=False, *create_using*=False)

```
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*=False)

```
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*=False)

```
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
```

18.2 Variables

Name	Description
__author__	**Value:** `'Aric Hagberg (hagberg@lanl.gov)\nDan Schult (d-schult@colgate.edu)'` *(type=**str**)*
__credits__	**Value:** `''` *(type=**str**)*
__date__	**Value:** `'$Date: 2005/03/30 23:56:28 $'` *(type=**str**)*
__revision__	**Value:** `'$Revision: 1.29 $'` *(type=**str**)*

19 Module NX.isomorph

```
Fast checking to see if graphs are not isomorphic.

This isn't a graph isomorphism checker.
```

19.1 Functions

fast_graph_could_be_isomorphic(*G1*, *G2*)

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

Checks for matching degree and triangle sequences.
```

faster_graph_could_be_isomorphic(*G1*, *G2*)

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

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

graph_could_be_isomorphic(*G1*, *G2*)

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

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

19.2 Variables

Name	Description
__author__	**Value:** 'Pieter Swart (swart@lanl.gov)\nDan Schult (dschult@colgate.edu)' *(type=str)*
__credits__	**Value:** '' *(type=str)*
__date__	**Value:** '$Date: 2005/03/30 23:56:28 $' *(type=str)*
__revision__	**Value:** '$Revision: 1.10 $' *(type=str)*

20 Module NX.operators

Operations on graphs; including union, complement, subgraph.

20.1 Functions

cartesian_product(*G*, *H*)

Return the Cartesian product of G and H.

Tested only on Graph class.

complement(*G*, *create_using*=None, ***kwds*)

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, ***kwds*)

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, ***kwds*)

```
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, ***kwds*)

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

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.
```

20.2 Variables

Name	Description
__author__	**Value:** 'Aric Hagberg (hagberg@lanl.gov)\nPieter Swart - (swart@lanl.gov)\nDan Schult(d... *(type=str)*
__credits__	**Value:** '' *(type=str)*
__date__	**Value:** '$Date: 2005/03/30 23:56:28 $' *(type=str)*
__revision__	**Value:** '$Revision: 1.24 $' *(type=str)*

21 Module NX.paths

Shortest path length, diameter, radius, eccentricity, and related methods.

21.1 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.

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

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

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

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.

21.2 Variables

Name	Description
__author__	**Value:** `'Aric Hagberg (hagberg@lanl.gov)\nDan Schult(dschult@colgate.edu)'` *(type=**str**)*
__credits__	**Value:** `''` *(type=**str**)*
__date__	**Value:** `'$Date: 2005/03/30 23:56:28 $'` *(type=**str**)*
__revision__	**Value:** `'$Revision: 1.29 $'` *(type=**str**)*

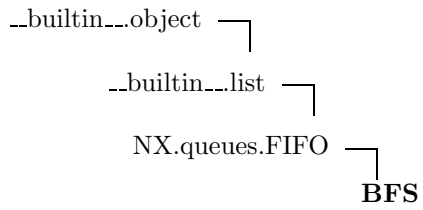
22 Module NX.queues

```
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())
```

22.1 Variables

Name	Description
__author__	**Value:** `'Aric Hagberg (hagberg@lanl.gov)'` *(type=str)*
__credits__	**Value:** `''` *(type=str)*
__date__	**Value:** `'$Date: 2005/03/30 23:56:28 $'` *(type=str)*
__revision__	**Value:** `'$Revision: 1.10 $'` *(type=str)*

22.2 Class BFS




```
Breadth first search queue
```

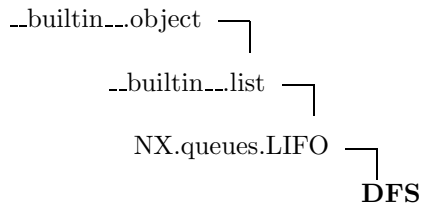
22.2.1 Methods

__init__(*self*)
Overrides: NX.queues.FIFO.__init__

update(*self*, *item*)

Inherited from FIFO: pop
Inherited from list: __add__, __contains__, __delitem__, __delslice__, __eq__, __ge__, __getattribute__, __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__

22.3 Class DFS




```
Depth first search queue
```

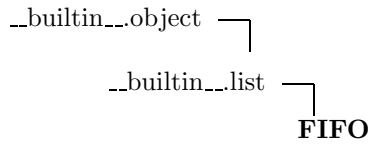
22.3.1 Methods

__init__(*self*)
Overrides: NX.queues.LIFO.__init__

update(*self*, *item*)

Inherited from list: __add__, __contains__, __delitem__, __delslice__, __eq__, __ge__, __getattribute__, __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__

22.4 Class FIFO



Known Subclasses: BFS

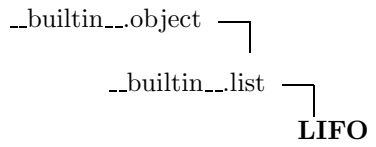
22.4.1 Methods

__init__(*self*)
Overrides: __builtin__.list.__init__

pop(*self*)
Overrides: __builtin__.list.pop

Inherited from list: __add__, __contains__, __delitem__, __delslice__, __eq__, __ge__, __getattribute__, __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__

22.5 Class LIFO



Known Subclasses: DFS

22.5.1 Methods

__init__(*self*)
Overrides: __builtin__.list.__init__

Inherited from list: __add__, __contains__, __delitem__, __delslice__, __eq__, __ge__, __getattribute__, __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__

22.6 Class Priority

22.6.1 Methods

__init__(*self*, *f*=<function <lambda> at 0x403f779c>)

__len__(*self*)

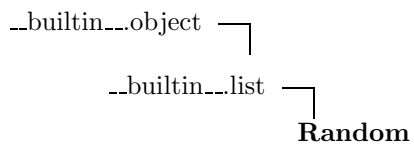
append(*self*, *item*)

extend(*self*, *items*)

pop(*self*)

smallest(*self*)

22.7 Class Random



Known Subclasses: RFS

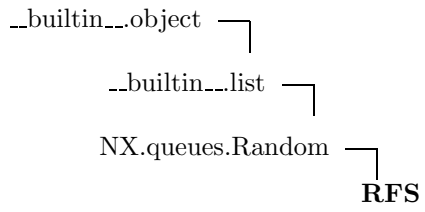
22.7.1 Methods

__init__(*self*)
Overrides: __builtin__.list.__init__

pop(*self*)
Overrides: __builtin__.list.pop

Inherited from list: __add__, __contains__, __delitem__, __delslice__, __eq__, __ge__, __getattribute__, __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__

22.8 Class RFS




```
Random search queue
```

22.8.1 Methods

__init__(*self*)
Overrides: NX.queues.Random.__init__

update(*self*, *item*)

Inherited from Random: pop
Inherited from list: __add__, __contains__, __delitem__, __delslice__, __eq__, __ge__, __getattribute__, __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__

23 Module NX.release

Release data for NetworkX.

23.1 Variables

Name	Description
authors	**Value:** {'Swart': ('Pieter Swart', 'swart@lanl.gov'), 'Schult': ('Dan Schult', 'dschu... *(type=dict)*
date	**Value:** 'Sun Apr 10 10:55:09 2005' *(type=str)*
description	**Value:** 'A package for creating and manipulating large graphs and networks.' *(type=str)*
keywords	**Value:** ['Networks', 'Graph Theory', 'Mathematics'] *(type=list)*
license	**Value:** 'LGPL' *(type=str)*
long_description	**Value:** '\nLong\n' *(type=str)*
name	**Value:** 'NX' *(type=str)*
platforms	**Value:** ['Linux', 'Mac OSX', 'Windows XP/2000/NT'] *(type=list)*
url	**Value:** 'http://networkx.sorceforge.net' *(type=str)*
version	**Value:** '0.1.1-7' *(type=str)*

24 Module NX.search

Search algorithms, shortest path, spanning trees, etc.

See also NX.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},
}
```

24.1 Functions

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

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.
```

is_connected(*G*)

`True if G is connected`

node_connected_component(*G*, *v*)

`Return the connected component to which v belongs as a list of nodes.`

number_connected_components(*G*)

`Return number of connected components of G.`

24.2 Variables

Name	Description
__author__	**Value:** `'Aric Hagberg (hagberg@lanl.gov)\nDan Schult(dschult@colgate.edu)'` *(type=`str`)*
__credits__	**Value:** `''` *(type=`str`)*
__date__	**Value:** `'$Date: 2005/04/09 00:28:57 $'` *(type=`str`)*
__revision__	**Value:** `'$Revision: 1.30 $'` *(type=`str`)*

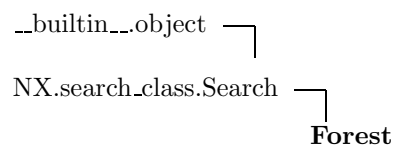
25 Module NX.search_class

```
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 lenghts, 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 NX.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},
}
```

25.1 Variables

Name	Description
__author__	**Value:** `'Aric Hagberg (hagberg@lanl.gov)'` *(type=**str**)*
__credits__	**Value:** `''` *(type=**str**)*
__date__	**Value:** `'$Date: 2005/03/30 23:56:28 $'` *(type=**str**)*
__revision__	**Value:** `'$Revision: 1.11 $'` *(type=**str**)*

25.2 Class Forest




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

25.2.1 Methods

__init__(*self*, *G*, *queue*=<class 'NX.queues.DFS'>, ***kwds*)
Overrides: NX.search_class.Search.__init__

end_tree(*self*, *v*)

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

Overrides: NX.search_class.Search.end_tree

lastseen_edge(*self*, *e*)

```
Visitor function called the last time an edge is encountered.
```

Overrides: NX.search_class.Search.lastseen_edge

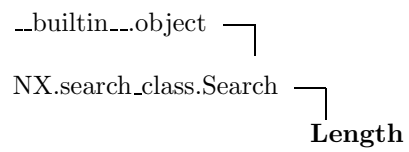
start_tree(*self*, *v*)

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

Overrides: NX.search_class.Search.start_tree

Inherited from Search: firstseen_edge, firstseen_vertex, lastseen_vertex, search
Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__

25.3 Class Length




```
Path length visitor.
Returns dictionary of path lengths from vertex v.
Useful especially in BFS (gives shortest paths).
```

25.3.1 Methods

__init__(*self*, *G*, *queue*=<class 'NX.queues.BFS'>, ***kwds*)
Overrides: NX.search_class.Search.__init__

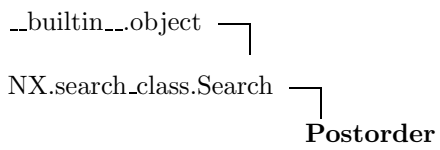
firstseen_edge(*self*, *e*)

```
Visitor function called the first time an edge is encountered.
```

Overrides: NX.search_class.Search.firstseen_edge

Inherited from Search: end_tree, firstseen_vertex, lastseen_edge, lastseen_vertex, search, start_tree
Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__

25.4 Class Postorder




```
Postorder visitor
Builds a list of nodes in postorder of search.
Returns a list of lists if the graph is not connected.
```

25.4.1 Methods

__init__(*self*, *G*, *queue*=<class 'NX.queues.DFS'>, ***kwds*)
Overrides: NX.search_class.Search.__init__

end_tree(*self*, *v*)

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

Overrides: NX.search_class.Search.end_tree

lastseen_vertex(*self*, *v*)

```
Visitor function called the last time a vertex is encountered.
```

Overrides: NX.search_class.Search.lastseen_vertex

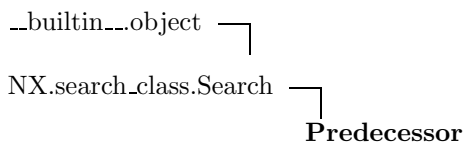
start_tree(*self*, *v*)

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

Overrides: NX.search_class.Search.start_tree

Inherited from Search: firstseen_edge, firstseen_vertex, lastseen_edge, search
Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__

25.5 Class Predecessor




```
Predeceessor visitor
Builds a dict of nodes with sucessor vertex list as data.
path method returns path lengths from source to target.
```

25.5.1 Methods

__init__(*self*, *G*, *queue*=<class 'NX.queues.DFS'>, ***kwds*)
Overrides: NX.search_class.Search.__init__

firstseen_vertex(*self*, *v*)

```
Visitor function called the first time a vertex is encountered.
```

Overrides: NX.search_class.Search.firstseen_vertex

lastseen_edge(*self*, *e*)

```
Visitor function called the last time an edge is encountered.
```

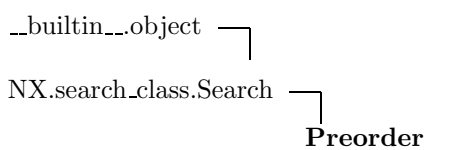
Overrides: NX.search_class.Search.lastseen_edge

path(*self*, *target*)

```
Gets one shortest path to target out of predecessor hash.
There might be more than one
```

Inherited from Search: end_tree, firstseen_edge, lastseen_vertex, search, start_tree
Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__

25.6 Class Preorder




```
Preorder visitor
Builds a list of nodes in preorder of search.
Returns a list of lists if the graph is not connected.
```

25.6.1 Methods

__init__(*self*, *G*, *queue*=<class 'NX.queues.DFS'>, ***kwds*)
Overrides: NX.search_class.Search.__init__

end_tree(*self*, *v*)

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

Overrides: NX.search_class.Search.end_tree

firstseen_vertex(*self*, *v*)

```
Visitor function called the first time a vertex is encountered.
```

Overrides: NX.search_class.Search.firstseen_vertex

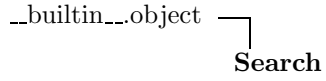
start_tree(*self*, *v*)

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

Overrides: NX.search_class.Search.start_tree

Inherited from Search: firstseen_edge, lastseen_edge, lastseen_vertex, search
Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__

25.7 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
```

25.7.1 Methods

__init__(*self*, *G*, *queue*=<class 'NX.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)

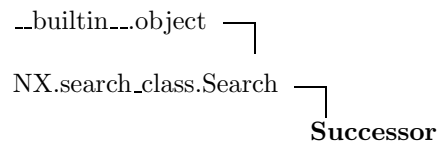
```
Search the graph.

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__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__

25.8 Class Successor




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

25.8.1 Methods

__init__(*self*, *G*, *queue*=<class 'NX.queues.DFS'>, ***kwds*)
Overrides: NX.search_class.Search.__init__

firstseen_vertex(*self*, *v*)

```
Visitor function called the first time a vertex is encountered.
```

Overrides: NX.search_class.Search.firstseen_vertex

lastseen_edge(*self*, *e*)

```
Visitor function called the last time an edge is encountered.
```

Overrides: NX.search_class.Search.lastseen_edge

Inherited from Search: end_tree, firstseen_edge, lastseen_vertex, search, start_tree
Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__

26 Module NX.spectrum

Laplacian, adjacency matrix, and spectrum of graphs.

Uses Numeric.

26.1 Functions

adj_matrix(G, u=None)

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

fan_chung_laplacian(G)

Return Fan Chung Laplacian of graph

laplacian(G)

Return standard Laplacian of graph

26.2 Variables

Name	Description
__author__	**Value:** 'Aric Hagberg (hagberg@lanl.gov)\nPieter Swart - (swart@lanl.gov)\nDan Schult(d... *(type=str)*
__credits__	**Value:** '' *(type=str)*
__date__	**Value:** '$Date: 2005/03/30 23:56:28 $' *(type=str)*
__revision__	**Value:** '$Revision: 1.16 $' *(type=str)*

27 Package NX.tests

27.1 Modules

28 Module NX.tests.test

28.1 Variables

Name	Description
__author__	**Value:** 'Aric Hagberg (hagberg@lanl.gov)\nPieter Swart - (swart@lanl.gov)' *(type=str)*
__credits__	**Value:** '' *(type=str)*
__date__	**Value:** '$Date: 2005/03/30 23:56:28 $' *(type=str)*
__revision__	**Value:** '$Revision: 1.24 $' *(type=str)*
mlist	**Value:** ['NX.base', 'NX.centrality', 'NX.cliques', 'NX.-cluster', 'NX.cores', 'NX.draw... *(type=list)*
runner	**Value:** <unittest.TextTestRunner object at 0x408f944c> *(type=TextTestRunner)*
suite	**Value:** <unittest.TestSuite tests=[<unittest.TestSuite-tests=[/nh/u/aric/networks/NX/... *(type=TestSuite)*

29 Module NX.tests.test2

29.1 Variables

Name	Description
files	**Value:** `[]` *(type=`list`)*

30 Module NX.utils

Utilities for NX package

30.1 Functions

discrete_sequence(*n*, ***kwds*)

Return sample sequence of length n from a given discrete distribution

distribution=histogram of values, will be normalized

gsl_pareto_sequence(*n*, ***kwds*)

Return sample sequence of length n from a Pareto distribution.

gsl_poisson_sequence(*n*, ***kwds*)

Return sample sequence of length n from a Poisson distribution.

gsl_powerlaw_sequence(*n*, ***kwds*)

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

gsl_uniform_sequence(*n*, ***kwds*)

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*, ***kwds*)

Return sample sequence of length n from a Pareto distribution.

powerlaw_sequence(*n*, ***kwds*)

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

scipy_discrete_sequence(*n*, ***kwds*)

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

distribution=histogram of values, will be normalized
```

scipy_pareto_sequence(*n*, ***kwds*)

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

scipy_poisson_sequence(*n*, ***kwds*)

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

scipy_powerlaw_sequence(*n*, ***kwds*)

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

scipy_uniform_sequence(*n*, ***kwds*)

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

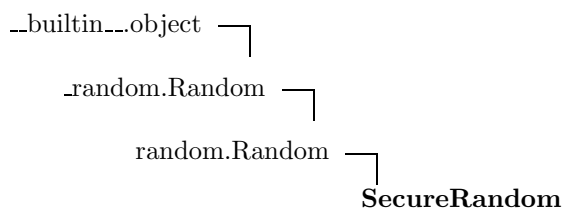
uniform_sequence(*n*, ***kwds*)

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

30.2 Variables

Name	Description
__author__	**Value:** `'Aric Hagberg (hagberg@lanl.gov)\nDan Schult(dschult@colgate.edu)'` *(type=`str`)*
__credits__	**Value:** `''` *(type=`str`)*
__date__	**Value:** `'$Date: 2005/03/30 23:56:28 $'` *(type=`str`)*
__revision__	**Value:** `'$Revision: 1.17 $'` *(type=`str`)*

30.3 Class SecureRandom




```
This class replaces the random number generator in Python
with /dev/random or /dev/urandom.  If you create an instance
and put it in random._inst then the usual Python random calls
will use /dev/random as the underlying random number generator.
```

30.3.1 Methods

__init__(*self*, *paranoid*=`False`)
Overrides: random.Random.__init__

getstate(*self*)

```
Return internal state; can be passed to setstate() later.
```

Overrides: random.Random.getstate extit(inherited documentation)

jumpahead(*self*, *ignore*)
Overrides: _random.Random.jumpahead

random(*self*)
Overrides: _random.Random.random

seed(*self*, *ignore*)
Overrides: random.Random.seed

setstate(*self*, *ignore*)
Overrides: random.Random.setstate

Inherited from object: __delattr__, __hash__, __reduce_ex__, __repr__, __setattr__, __str__
Inherited from Random: __getstate__, __reduce__, __setstate__, betavariate, choice, expovariate, gammavariate, gauss, lognormvariate, normalvariate, paretovariate, randint, randrange, sample, shuffle, uniform, vonmisesvariate, weibullvariate

30.3.2 Class Variables

Name	Description
Inherited from Random: VERSION *(p. ??)*	

31 Module NX.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,... 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 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.

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)

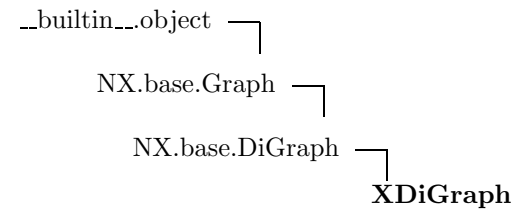
G.has_edge(n1,n2), G.has_neighbor(n1,n2)

```
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 NX 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
__author__	**Value:** 'Aric Hagberg (hagberg@lanl.gov)\nPieter Swart - (swart@lanl.gov)\nDan Schult(d... *(type=str)*

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*, ***kwds*)

```
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: NX.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 arbirary 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: NX.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: NX.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 multiededges. Each node and each edge in original
graph are added to the copy.
```

Overrides: NX.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: NX.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: NX.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: NX.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: NX.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: NX.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: NX.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: NX.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: NX.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: NX.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: NX.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: NX.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: NX.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: NX.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 garantee which one it is.
```

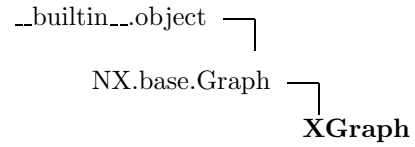
Overrides: NX.base.DiGraph.to_undirected

Inherited from DiGraph: __getitem__, add_node, add_nodes_from, clear, degree_iter, delete_node, delete_nodes_from, in_degree_iter, is_directed, out_degree_iter, predecessors_iter, 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
Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__

31.2.2 Class Variables

Name	Description
Inherited from Graph: __slotnames__ *(p. 22)*	

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

__init__(*self*, ***kwds*)

```
Initialize XGraph.

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: NX.base.Graph.__init__

add_cycle(*self*, *nlist*)

```
Add the cycle of nodes in nlist to graph
```

Overrides: NX.base.Graph.add_cycle

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 arbirary 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: NX.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: NX.base.Graph.add_edges_from

add_path(*self*, *nlist*)

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

Overrides: NX.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 multiededges. Each node and each edge in original
graph are added to the copy.
```

Overrides: NX.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: NX.base.Graph.degree

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

```
This is the degree() method returned in interator form.
If with_labels=True, iterator yields 2-tuples of form (n,degree(n))
(like iteritems() on a dict.)
```

Overrides: NX.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: NX.base.Graph.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).  The container must be iterable or an iterator, and
is iterated over once. Edges that are not in the graph are ignored.
```

Overrides: NX.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: NX.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: NX.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: NX.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: NX.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: NX.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: NX.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: NX.base.Graph.neighbors

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

`Return an iterator for neighbors of n.`

Overrides: NX.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: NX.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: NX.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: NX.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: NX.base.Graph.to_directed

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

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

31.3.2 Class Variables

Name	Description
Inherited from Graph: __slotnames__ *(p. 22)*	

Index