Software for Complex Networks

Release: 3.3
Date: Apr 06, 2024

NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. It provides:

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

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

Citing

To cite NetworkX please use the following publication:

Aric A. Hagberg, Daniel A. Schult and Pieter J. Swart, "Exploring network structure, dynamics, and function using NetworkX", in Proceedings of the 7th Python in Science Conference (SciPy2008), Gäel Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11–15, Aug 2008

PDF BibTeX

Audience

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

Python

Python is a powerful programming language that allows simple and flexible representations of networks as well as clear and concise expressions of network algorithms. Python has a vibrant and growing ecosystem of packages that NetworkX uses to provide more features such as numerical linear algebra and drawing. In order to make the most out of NetworkX you will want to know how to write basic programs in Python. Among the many guides to Python, we recommend the Python documentation and the text by Alex Martelli [Martelli03].

License

NetworkX is distributed with the 3-clause BSD license.


```
Copyright (C) 2004-2024, NetworkX Developers
Aric Hagberg <hagberg@lanl.gov>
Dan Schult <dschult@colgate.edu>
Pieter Swart <swart@lanl.gov>
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:

  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

  * Redistributions in binary form must reproduce the above
    copyright notice, this list of conditions and the following
    disclaimer in the documentation and/or other materials provided
    with the distribution.

  * Neither the name of the NetworkX Developers nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```


Bibliography

Skip to main content

[BA02] R. Albert and A.-L. Barabási, “Statistical mechanics of complex networks”, Reviews of Modern Physics, 74, pp. 47-97, 2002. https://arxiv.org/abs/cond-mat/0106096

[Bollobas01] B. Bollobás, “Random Graphs”, Second Edition, Cambridge University Press, 2001.

[BE05] U. Brandes and T. Erlebach, “Network Analysis: Methodological Foundations”, Lecture Notes in Computer Science, Volume 3418, Springer-Verlag, 2005.

[Diestel97] R. Diestel, “Graph Theory”, Springer-Verlag, 1997. http://diestel-graph-theory.com/index.html

[DM03] S.N. Dorogovtsev and J.F.F. Mendes, “Evolution of Networks”, Oxford University Press, 2003.

[Martelli03] A. Martelli, “Python in a Nutshell”, O’Reilly Media Inc, 2003.

[Newman03] M.E.J. Newman, “The Structure and Function of Complex Networks”, SIAM Review, 45, pp. 167-256, 2003. http://epubs.siam.org/doi/abs/10.1137/S003614450342480

[Sedgewick02] R. Sedgewick, “Algorithms in C: Parts 1-4: Fundamentals, Data Structure, Sorting, Searching”, Addison Wesley Professional, 3rd ed., 2002.

© Copyright 2004-2024, NetworkX Developers.

Built with the PyData Sphinx Theme 0.15.2.

Created using Sphinx 7.2.6.

Install

NetworkX requires Python 3.10, 3.11, or 3.12. If you do not already have a Python environment configured on your computer, please see the instructions for installing the full scientific Python stack.

Below we assume you have the default Python environment already configured on your computer and you intend to install `networkx` inside of it. If you want to create and work with Python virtual environments, please follow instructions on venv and virtual environments.

First, make sure you have the latest version of `pip` (the Python package manager) installed. If you do not, refer to the Pip documentation and install `pip` first.

Install the released version

Install the current release of `networkx` with `pip`:

```
$ pip install networkx[default]
```

To upgrade to a newer release use the `--upgrade` flag:

```
$ pip install --upgrade networkx[default]
```

If you do not have permission to install software systemwide, you can install into your user directory using the `--user` flag:

```
$ pip install --user networkx[default]
```

If you do not want to install our dependencies (e.g., `numpy`, `scipy`, etc.), you can use:

```
$ pip install networkx
```

This may be helpful if you are using PyPy or you are working on a project that only needs a limited subset of our functionality and you want to limit the number of dependencies.

Alternatively, you can manually download `networkx` from GitHub or PyPI. To install one of these versions,

```
$ pip install .[default]
```

Install the development version

If you have Git installed on your system, it is also possible to install the development version of `networkx`.

Before installing the development version, you may need to uninstall the standard version of `networkx` using `pip`:

```
$ pip uninstall networkx
```

Then do:

```
$ git clone https://github.com/networkx/networkx.git
$ cd networkx
$ pip install -e .[default]
```

The `pip install -e .[default]` command allows you to follow the development branch as it changes by creating links in the right places and installing the command line scripts to the appropriate locations.

Then, if you want to update `networkx` at any time, in the same directory do:

```
$ git pull
```

Extra packages

Note

Some optional packages may require compiling C or C++ code. If you have difficulty installing these packages with `pip`, please consult the homepages of those packages.

The following extra packages provide additional functionality. See the files in the `requirements/` directory for information about specific version requirements.

- PyGraphviz and pydot provide graph drawing and graph layout algorithms via GraphViz.
- lxml used for GraphML XML format.

To install `networkx` and extra packages, do:

Skip to main content

```
$ pip install networkx[default,extra]
```

To explicitly install all optional packages, do:

```
$ pip install pygraphviz pydot lxml
```

Or, install any optional package (e.g., `pygraphviz`) individually:

```
$ pip install pygraphviz
```

Testing

NetworkX uses the Python `pytest` testing package. You can learn more about pytest on their homepage.

Test a source distribution

You can test the complete package from the unpacked source directory with:

```
pytest networkx
```

Test an installed package

From a shell command prompt you can test the installed package with:

```
pytest --pyargs networkx
```

© Copyright 2004-2024, NetworkX Developers.

Created using Sphinx 7.2.6.

Built with the PyData Sphinx Theme 0.15.2.

Tutorial

This guide can help you start working with NetworkX.

Creating a graph

Create an empty graph with no nodes and no edges.

```
import networkx as nx
G = nx.Graph()
```

By definition, a `Graph` is a collection of nodes (vertices) along with identified pairs of nodes (called edges, links, etc). In NetworkX, nodes can be any hashable object e.g., a text string, an image, an XML object, another Graph, a customized node object, etc.

Note

Python's `None` object is not allowed to be used as a node. It determines whether optional function arguments have been assigned in many functions.

Nodes

The graph `G` can be grown in several ways. NetworkX includes many graph generator functions and facilities to read and write graphs in many formats. To get started though we'll look at simple manipulations. You can add one node at a time,

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

or add nodes from any iterable container, such as a list

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

You can also add nodes along with node attributes if your container yields 2-tuples of the form `(node, node_attribute_dict)`:

```
G.add_nodes_from([(4, {"color": "red"}), (5, {"color": "green"})])
```

Node attributes are discussed further below.

Nodes from one graph can be incorporated into another:

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

`G` now contains the nodes of `H` as nodes of `G`. In contrast, you could use the graph `H` as a node in `G`.

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

The graph `G` now contains `H` as a node. This flexibility is very powerful as it allows graphs of graphs, graphs of files, graphs of functions and much more. It is worth thinking about how to structure your application so that the nodes are useful entities. Of course you can always use a unique identifier in `G` and have a separate dictionary keyed by identifier to the node information if you prefer.

Note

You should not change the node object if the hash depends on its contents.

Edges

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

```
G.add_edge(1, 2)
e = (2, 3)
G.add_edge(*e)  # unpack edge tuple*
```

by adding a list of edges,

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

or by adding any ebunch of edges. An *ebunch* is any iterable container of edge-tuples. An edge-tuple can be a 2-tuple of nodes or a 3-tuple with 2 nodes followed by an edge attribute dictionary, e.g., `(2, 3, {'weight': 3.1415})`. Edge attributes are discussed further below.

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

Skip to main content

There are no complaints when adding existing nodes or edges. For example, after removing all nodes and edges,

```
G.clear()
```

we add new nodes/edges and NetworkX quietly ignores any that are already present.

```
G.add_edges_from([(1, 2), (1, 3)])
G.add_node(1)
G.add_edge(1, 2)
G.add_node("spam")        # adds node "spam"
G.add_nodes_from("spam")  # adds 4 nodes: 's', 'p', 'a', 'm'
G.add_edge(3, 'm')
```

At this stage the graph `G` consists of 8 nodes and 3 edges, as can be seen by:

```
G.number_of_nodes()
```

```
8
```

```
G.number_of_edges()
```

```
3
```

Note

The order of adjacency reporting (e.g., `G.adj`, `G.successors`, `G.predecessors`) is the order of edge addition. However, the order of G.edges is the order of the adjacencies which includes both the order of the nodes and each node's adjacencies. See example below:

```
DG = nx.DiGraph()
DG.add_edge(2, 1)   # adds the nodes in order 2, 1
DG.add_edge(1, 3)
DG.add_edge(2, 4)
DG.add_edge(1, 2)
assert list(DG.successors(2)) == [1, 4]
assert list(DG.edges) == [(2, 1), (2, 4), (1, 3), (1, 2)]
```

Examining elements of a graph

Skip to main content

We can examine the nodes and edges. Four basic graph properties facilitate reporting: `G.nodes`, `G.edges`, `G.adj` and `G.degree`. These are set-like views of the nodes, edges, neighbors (adjacencies), and degrees of nodes in a graph. They offer a continually updated read-only view into the graph structure. They are also dict-like in that you can look up node and edge data attributes via the views and iterate with data attributes using methods `.items()`, `.data()`. If you want a specific container type instead of a view, you can specify one. Here we use lists, though sets, dicts, tuples and other containers may be better in other contexts.

```
list(G.nodes)
```

```
[1, 2, 3, 'spam', 's', 'p', 'a', 'm']
```

```
list(G.edges)
```

```
[(1, 2), (1, 3), (3, 'm')]
```

```
list(G.adj[1])  # or list(G.neighbors(1))
```

```
[2, 3]
```

```
G.degree[1]  # the number of edges incident to 1
```

```
2
```

One can specify to report the edges and degree from a subset of all nodes using an nbunch. An *nbunch* is any of: `None` (meaning all nodes), a node, or an iterable container of nodes that is not itself a node in the graph.

```
G.edges([2, 'm'])
```

```
EdgeDataView([(2, 1), ('m', 3)])
```

```
G.degree([2, 3])
```

```
DegreeView({2: 1, 3: 2})
```

Skip to main content

Removing elements from a graph

One can remove nodes and edges from the graph in a similar fashion to adding. Use methods `Graph.remove_node()`, `Graph.remove_nodes_from()`, `Graph.remove_edge()` and `Graph.remove_edges_from()`, e.g.

```
G.remove_node(2)
G.remove_nodes_from("spam")
list(G.nodes)
```

```
[1, 3, 'spam']
```

```
G.remove_edge(1, 3)
list(G)
```

```
[1, 3, 'spam']
```

Using the graph constructors

Graph objects do not have to be built up incrementally - data specifying graph structure can be passed directly to the constructors of the various graph classes. When creating a graph structure by instantiating one of the graph classes you can specify data in several formats.

```
G.add_edge(1, 2)
H = nx.DiGraph(G)  # create a DiGraph using the connections from G
list(H.edges())
```

```
[(1, 2), (2, 1)]
```

```
edgelist = [(0, 1), (1, 2), (2, 3)]
H = nx.Graph(edgelist)  # create a graph from an edge list
list(H.edges())
```

```
[(0, 1), (1, 2), (2, 3)]
```

```
adjacency_dict = {0: (1, 2), 1: (0, 2), 2: (0, 1)}
H = nx.Graph(adjacency_dict)  # create a Graph dict mapping nodes to nbrs
```

Skip to main content

```
[(0, 1), (0, 2), (1, 2)]
```

What to use as nodes and edges

You might notice that nodes and edges are not specified as NetworkX objects. This leaves you free to use meaningful items as nodes and edges. The most common choices are numbers or strings, but a node can be any hashable object (except `None`), and an edge can be associated with any object `x` using `G.add_edge(n1, n2, object=x)`.

As an example, `n1` and `n2` could be protein objects from the RCSB Protein Data Bank, and `x` could refer to an XML record of publications detailing experimental observations of their interaction.

We have found this power quite useful, but its abuse can lead to surprising behavior unless one is familiar with Python. If in doubt, consider using `convert_node_labels_to_integers()` to obtain a more traditional graph with integer labels.

Accessing edges and neighbors

In addition to the views `Graph.edges`, and `Graph.adj`, access to edges and neighbors is possible using subscript notation.

```
G = nx.Graph([(1, 2, {"color": "yellow"})])
G[1]  # same as G.adj[1]
```

```
AtlasView({2: {'color': 'yellow'}})
```

```
G[1][2]
```

```
{'color': 'yellow'}
```

```
G.edges[1, 2]
```

```
{'color': 'yellow'}
```

You can get/set the attributes of an edge using subscript notation if the edge already exists.

Skip to main content

```
G.add_edge(1, 3)
G[1][3]['color'] = "blue"
G.edges[1, 2]['color'] = "red"
G.edges[1, 2]
```

```
{'color': 'red'}
```

Fast examination of all (node, adjacency) pairs is achieved using `G.adjacency()`, or `G.adj.items()`. Note that for undirected graphs, adjacency iteration sees each edge twice.

```
FG = nx.Graph()
FG.add_weighted_edges_from([(1, 2, 0.125), (1, 3, 0.75), (2, 4, 1.2), (3, 4, 0.375)])
for n, nbrs in FG.adj.items():
   for nbr, eattr in nbrs.items():
       wt = eattr['weight']
       if wt < 0.5: print(f"({n}, {nbr}, {wt:.3})")
```

```
(1, 2, 0.125)
(2, 1, 0.125)
(3, 4, 0.375)
(4, 3, 0.375)
```

Convenient access to all edges is achieved with the edges property.

```
for (u, v, wt) in FG.edges.data('weight'):
    if wt < 0.5:
        print(f"({u}, {v}, {wt:.3})")
```

```
(1, 2, 0.125)
(3, 4, 0.375)
```

Adding attributes to graphs, nodes, and edges

Attributes such as weights, labels, colors, or whatever Python object you like, can be attached to graphs, nodes, or edges.

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

Graph attributes

Skip to main content

Assign graph attributes when creating a new graph

```
G = nx.Graph(day="Friday")
G.graph
```

```
{'day': 'Friday'}
```

Or you can modify attributes later

```
G.graph['day'] = "Monday"
G.graph
```

```
{'day': 'Monday'}
```

Node attributes

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

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

```
{'time': '5pm'}
```

```
G.nodes[1]['room'] = 714
G.nodes.data()
```

```
NodeDataView({1: {'time': '5pm', 'room': 714}, 3: {'time': '2pm'}})
```

Note that adding a node to `G.nodes` does not add it to the graph, use `G.add_node()` to add new nodes. Similarly for edges.

Edge Attributes

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

```
G.add_edge(1, 2, weight=4.7 )
G.add_edges_from([(3, 4), (4, 5)], color='red')
```

Skip to main content

```
G[1][2]['weight'] = 4.7
G.edges[3, 4]['weight'] = 4.2
```

The special attribute `weight` should be numeric as it is used by algorithms requiring weighted edges.

Directed graphs

The **`DiGraph`** class provides additional methods and properties specific to directed edges, e.g., **`DiGraph.out_edges`**, **`DiGraph.in_degree`**, **`DiGraph.predecessors()`**, **`DiGraph.successors()`** etc. To allow algorithms to work with both classes easily, the directed versions of **`neighbors`** is equivalent to **`successors`** while **`DiGraph.degree`** reports the sum of **`DiGraph.in_degree`** and **`DiGraph.out_degree`** even though that may feel inconsistent at times.

```
DG = nx.DiGraph()
DG.add_weighted_edges_from([(1, 2, 0.5), (3, 1, 0.75)])
DG.out_degree(1, weight='weight')
```

```
0.5
```

```
DG.degree(1, weight='weight')
```

```
1.25
```

```
list(DG.successors(1))
```

```
[2]
```

```
list(DG.neighbors(1))
```

```
[2]
```

Some algorithms work only for directed graphs and others are not well defined for directed graphs. Indeed the tendency to lump directed and undirected graphs together is dangerous. If you want to treat a directed graph as undirected for some measurement you should probably convert it using **`Graph.to_undirected()`** or with

```
H = nx.Graph(G)  # create an undirected graph H from a directed graph G
```

Skip to main content

Multigraphs

NetworkX provides classes for graphs which allow multiple edges between any pair of nodes. The `MultiGraph` and `MultiDiGraph` classes allow you to add the same edge twice, possibly with different edge data. This can be powerful for some applications, but many algorithms are not well defined on such graphs. Where results are well defined, e.g., `MultiGraph.degree()` we provide the function. Otherwise you should convert to a standard graph in a way that makes the measurement well defined.

```
MG = nx.MultiGraph()
MG.add_weighted_edges_from([(1, 2, 0.5), (1, 2, 0.75), (2, 3, 0.5)])
dict(MG.degree(weight='weight'))
```

```
{1: 1.25, 2: 1.75, 3: 0.5}
```

```
GG = nx.Graph()
for n, nbrs in MG.adjacency():
   for nbr, edict in nbrs.items():
       minvalue = min([d['weight'] for d in edict.values()])
       GG.add_edge(n, nbr, weight = minvalue)

nx.shortest_path(GG, 1, 3)
```

```
[1, 2, 3]
```

Graph generators and graph operations

In addition to constructing graphs node-by-node or edge-by-edge, they can also be generated by

1. Applying classic graph operations, such as:

`subgraph`(G, nbunch)	Returns the subgraph induced on nodes in nbunch.
`union`(G, H[, rename])	Combine graphs G and H.
`disjoint_union`(G, H)	Combine graphs G and H.
`cartesian_product`(G, H)	Returns the Cartesian product of G and H.
`compose`(G, H)	Compose graph G with H by combining nodes and edges into a single graph

Skip to main content

`complement`(G)	Returns the graph complement of G.
`create_empty_copy`(G[, with_data])	Returns a copy of the graph G with all of the edges removed.
`to_undirected`(graph)	Returns an undirected view of the graph `graph`.
`to_directed`(graph)	Returns a directed view of the graph `graph`.

2. Using a call to one of the classic small graphs, e.g.,

`petersen_graph`([create_using])	Returns the Petersen graph.
`tutte_graph`([create_using])	Returns the Tutte graph.
`sedgewick_maze_graph`([create_using])	Return a small maze with a cycle.
`tetrahedral_graph`([create_using])	Returns the 3-regular Platonic Tetrahedral graph.

3. Using a (constructive) generator for a classic graph, e.g.,

`complete_graph`(n[, create_using])	Return the complete graph `K_n` with n nodes.
`complete_bipartite_graph`(n1, n2[, create_using])	Returns the complete bipartite graph `K_{n_1,n_2}`.
`barbell_graph`(m1, m2[, create_using])	Returns the Barbell Graph: two complete graphs connected by a path.
`lollipop_graph`(m, n[, create_using])	Returns the Lollipop Graph; `K_m` connected to `P_n`.

like so:

```
K_5 = nx.complete_graph(5)
K_3_5 = nx.complete_bipartite_graph(3, 5)
barbell = nx.barbell_graph(10, 10)
lollipop = nx.lollipop_graph(10, 20)
```

4. Using a stochastic graph generator, e.g,

`erdos_renyi_graph`(n, p[, seed, directed])	Returns a $G_{n,p}$ random graph, also known as an Erdős-

Skip to main content

`watts_strogatz_graph`(n, k, p[, seed])	Returns a Watts–Strogatz small-world graph.
`barabasi_albert_graph`(n, m[, seed, ...])	Returns a random graph using Barabási–Albert preferential attachment
`random_lobster`(n, p1, p2[, seed])	Returns a random lobster graph.

like so:

```
er = nx.erdos_renyi_graph(100, 0.15)
ws = nx.watts_strogatz_graph(30, 3, 0.1)
ba = nx.barabasi_albert_graph(100, 5)
red = nx.random_lobster(100, 0.9, 0.9)
```

5. Reading a graph stored in a file using common graph formats

NetworkX supports many popular formats, such as edge lists, adjacency lists, GML, GraphML, LEDA and others.

```
nx.write_gml(red, "path.to.file")
mygraph = nx.read_gml("path.to.file")
```

For details on graph formats see Reading and writing graphs and for graph generator functions see Graph generators

Analyzing graphs

The structure of `G` can be analyzed using various graph-theoretic functions such as:

```
G = nx.Graph()
G.add_edges_from([(1, 2), (1, 3)])
G.add_node("spam")       # adds node "spam"
list(nx.connected_components(G))
```

```
[{1, 2, 3}, {'spam'}]
```

```
sorted(d for n, d in G.degree())
```

```
[0, 1, 1, 2]
```

Skip to main content

```
nx.clustering(G)
```

```
{1: 0, 2: 0, 3: 0, 'spam': 0}
```

Some functions with large output iterate over (node, value) 2-tuples. These are easily stored in a `dict` structure if you desire.

```
sp = dict(nx.all_pairs_shortest_path(G))
sp[3]
```

```
{3: [3], 1: [3, 1], 2: [3, 1, 2]}
```

See Algorithms for details on graph algorithms supported.

Drawing graphs

NetworkX is not primarily a graph drawing package but basic drawing with Matplotlib as well as an interface to use the open source Graphviz software package are included. These are part of the networkx.drawing module and will be imported if possible.

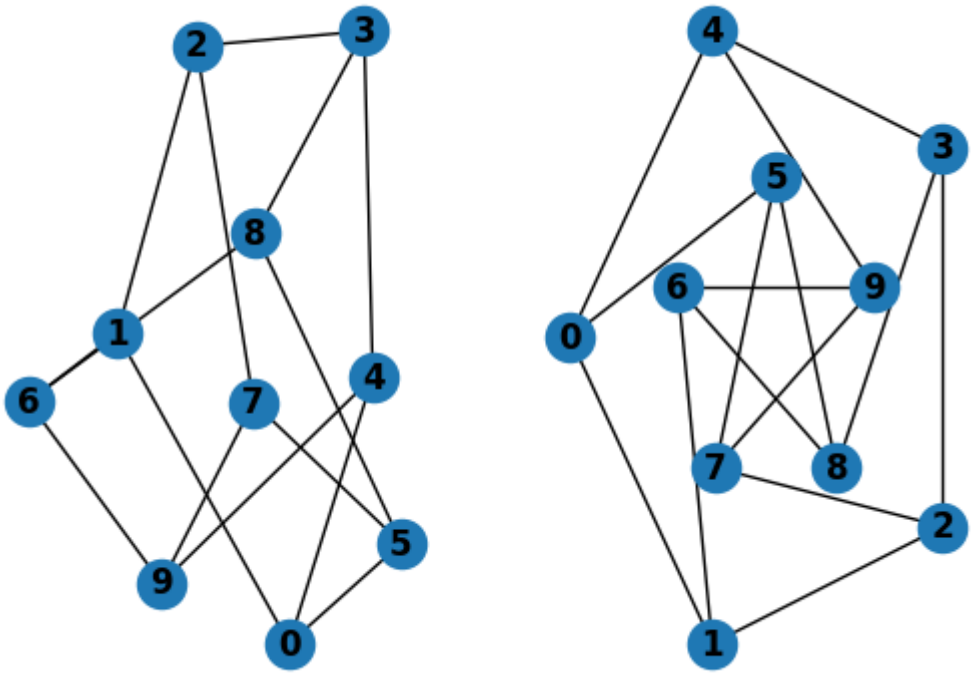
First import Matplotlib's plot interface (pylab works too)

```
import matplotlib.pyplot as plt
```

To test if the import of `~networkx.drawing.nx_pylab` was successful draw `G` using one of

```
G = nx.petersen_graph()
subax1 = plt.subplot(121)
nx.draw(G, with_labels=True, font_weight='bold')
subax2 = plt.subplot(122)
nx.draw_shell(G, nlist=[range(5, 10), range(5)], with_labels=True, font_weight='bold')
```

Skip to main content



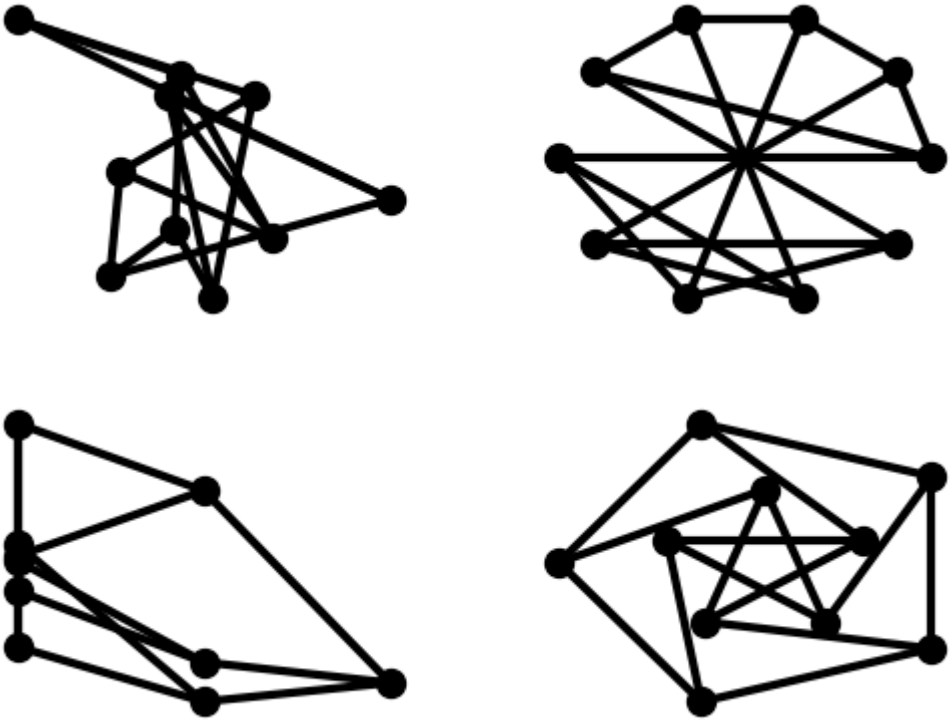
when drawing to an interactive display. Note that you may need to issue a Matplotlib

```
plt.show()
```

command if you are not using matplotlib in interactive mode.

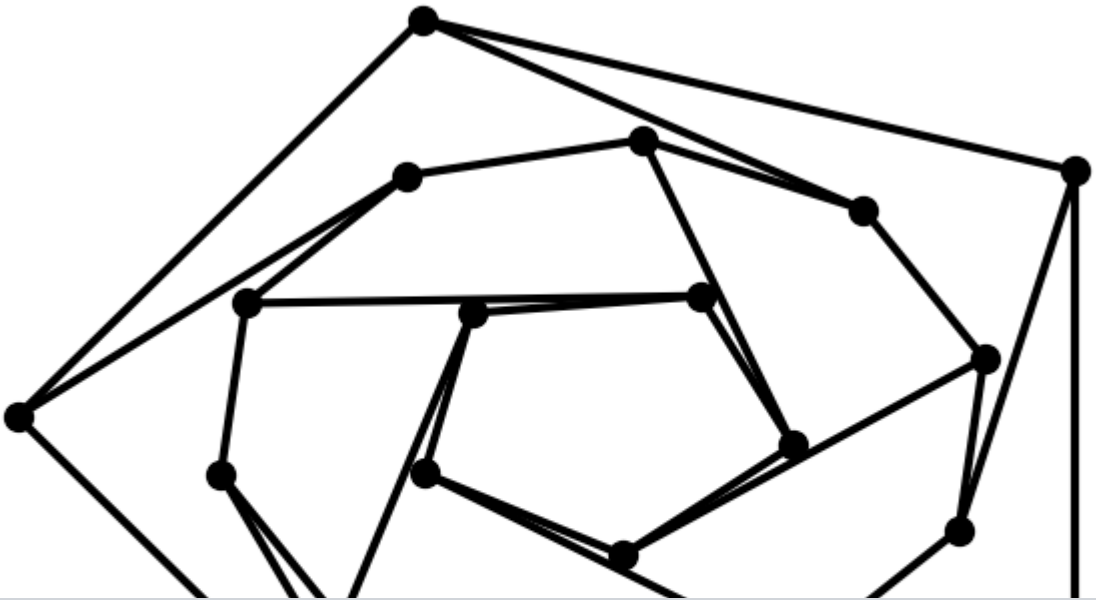
```
options = {
    'node_color': 'black',
    'node_size': 100,
    'width': 3,
}
subax1 = plt.subplot(221)
nx.draw_random(G, **options)
subax2 = plt.subplot(222)
nx.draw_circular(G, **options)
subax3 = plt.subplot(223)
nx.draw_spectral(G, **options)
subax4 = plt.subplot(224)
nx.draw_shell(G, nlist=[range(5,10), range(5)], **options)
```

Skip to main content



You can find additional options via `draw_networkx()` and layouts via the `layout module`. You can use multiple shells with `draw_shell()`.

```
G = nx.dodecahedral_graph()
shells = [[2, 3, 4, 5, 6], [8, 1, 0, 19, 18, 17, 16, 15, 14, 7], [9, 10, 11, 12, 13]]
nx.draw_shell(G, nlist=shells, **options)
```



© Copyright 2004-2024, NetworkX Developers.

Built with the PyData Sphinx Theme 0.15.2.

Created using Sphinx 7.2.6.

Reference

Release: 3.3
Date: Apr 06, 2024

Skip to main content

Skip to main content

Skip to main content

Skip to main content

© Copyright 2004-2024, NetworkX Developers.

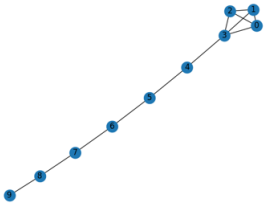
Created using Sphinx 7.2.6.

Built with the PyData Sphinx Theme 0.15.2.

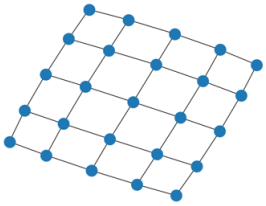
Gallery

General-purpose and introductory examples for NetworkX. The tutorial introduces conventions and basic graph manipulations.

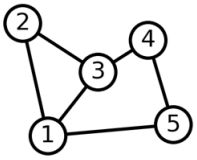
Basic



Properties

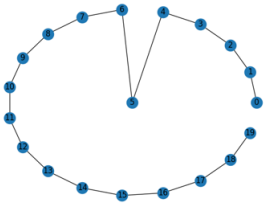


Read and write graphs.

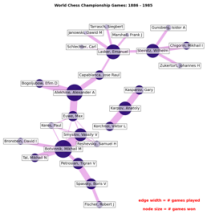


Simple graph

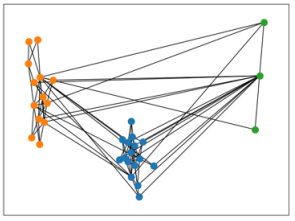
Drawing



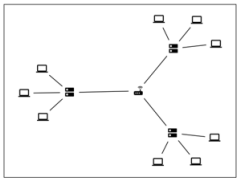
Custom Node Position



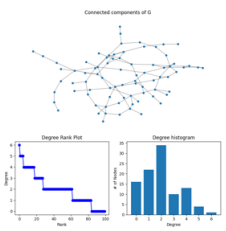
Chess Masters



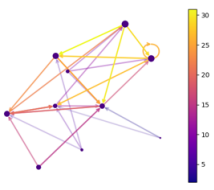
Cluster Layout



Custom node icons

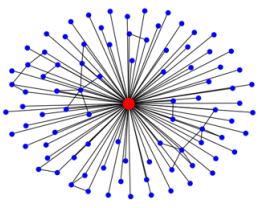


Degree Analysis



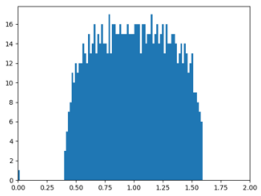
Directed Graph

Edge Colormap

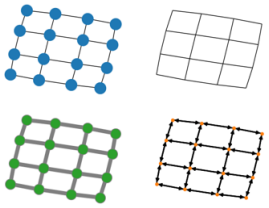


Ego Graph

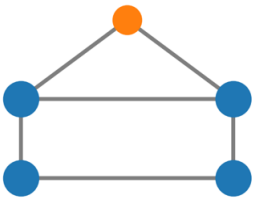
Skip to main content



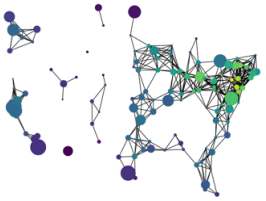
Eigenvalues



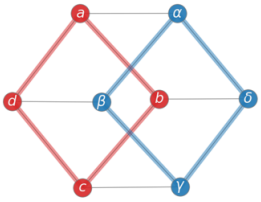
Four Grids



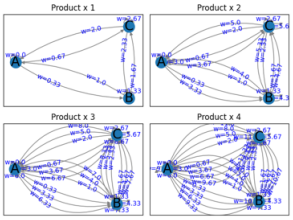
House With Colors



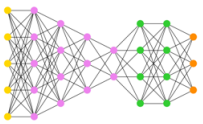
Knuth Miles



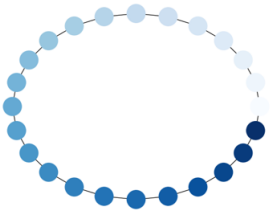
Labels And Colors



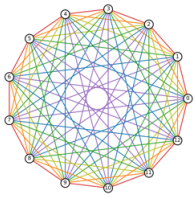
Plotting MultiDiGraph Edges and Labels



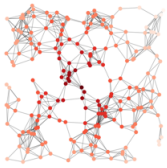
Multipartite Layout



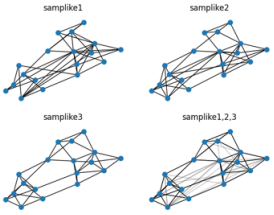
Node Colormap



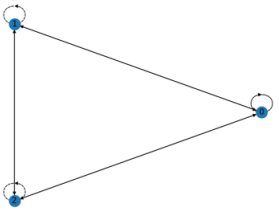
Rainbow Coloring



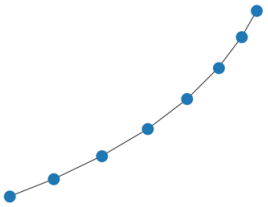
Random Geometric Graph



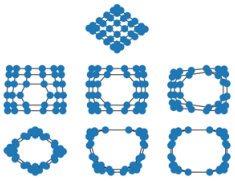
Sampson



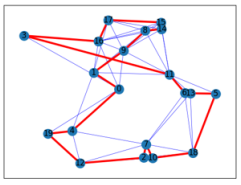
Self-loops



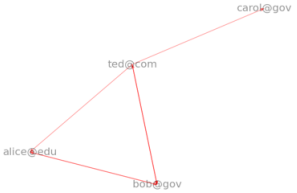
Simple Path



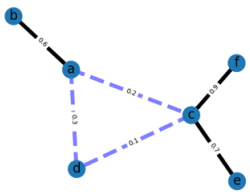
Spectral Embedding



Traveling Salesman Problem



Unix Email

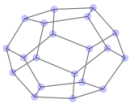


Weighted Graph

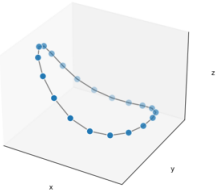
3D Drawing

Skip to main content

Mayavi2



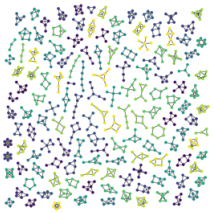
Animations of 3D rotation and random walk



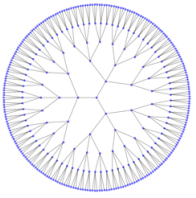
Basic matplotlib

Graphviz Layout

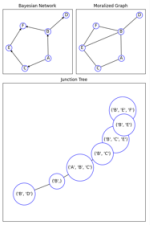
Examples using Graphviz layouts with `nx_pylab` for drawing. These examples need Graphviz and PyGraphviz.



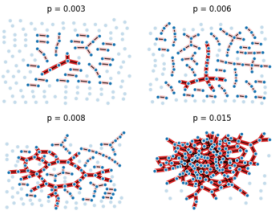
Atlas



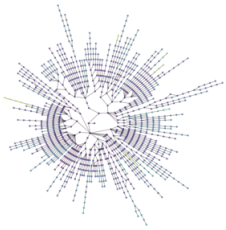
Circular Tree



Decomposition



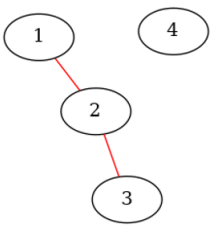
Giant Component



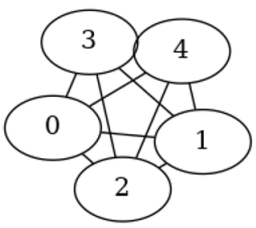
Lanl Routes

Graphviz Drawing

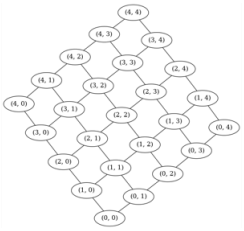
Examples using Graphviz for layout and drawing via `nx_agraph`. These examples need Graphviz and PyGraphviz.



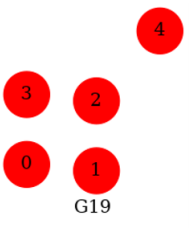
Attributes



Conversion



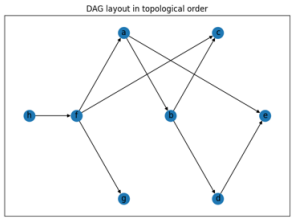
2D Grid



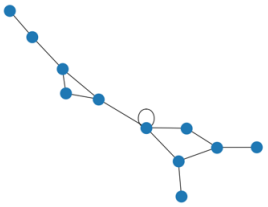
Atlas

Skip to main content

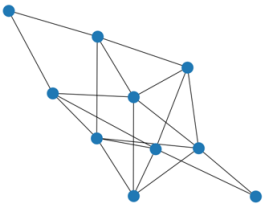
Graph



DAG - Topological Layout

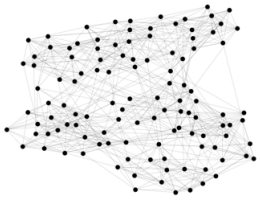


Degree Sequence

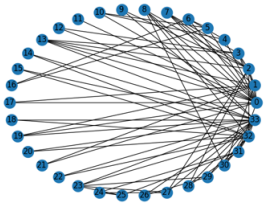


Erdos Renyi

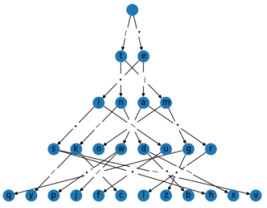
Expected Degree Sequence



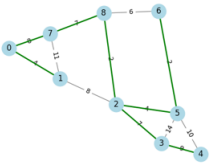
Football



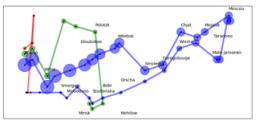
Karate Club



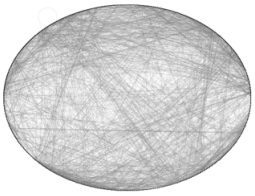
Morse Trie



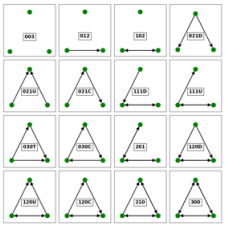
Minimum Spanning Tree



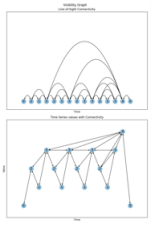
Napoleon Russian Campaign



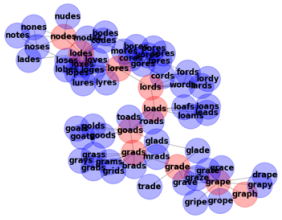
Roget



Triads

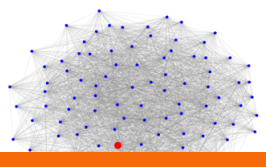


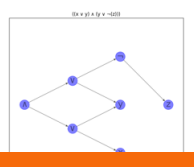
Visibility Graph



Words/Ladder Graph

Algorithms





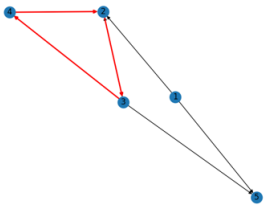
Skip to main content

Beam Search

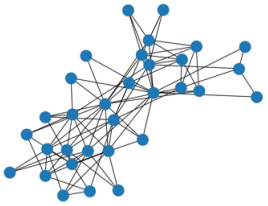
Betweenness Centrality

Blockmodel

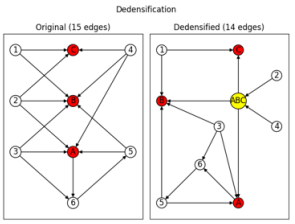
Circuits



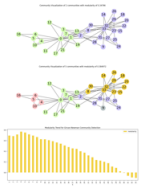
Cycle Detection



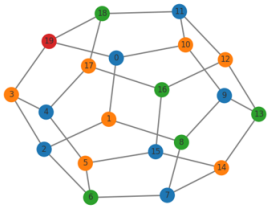
Davis Club



Dedensification



Community Detection using Girvan-Newman



Greedy Coloring

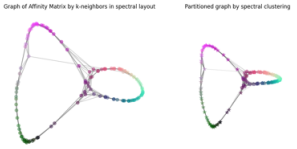
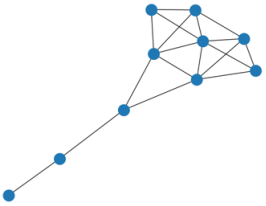
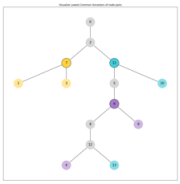


Image Segmentation via Spectral Graph Partitioning

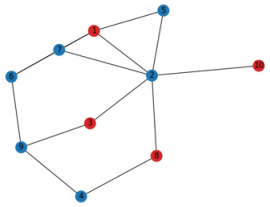
Iterated Dynamical Systems



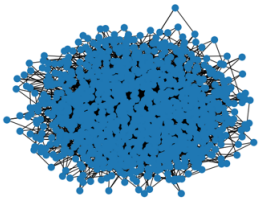
Krackhardt Centrality



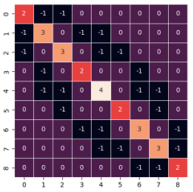
Lowest Common Ancestors



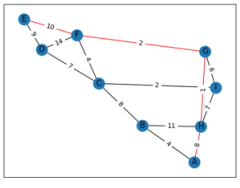
Maximum Independent Set



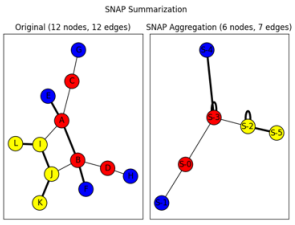
Parallel Betweenness



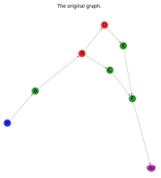
Reverse Cuthill--McKee



Find Shortest Path



SNAP Graph Summary



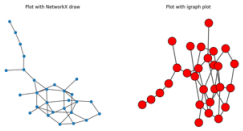
Subgraphs

External libraries

Examples of using NetworkX with external libraries.

Skip to main content

JavaScript

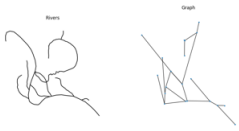


igraph

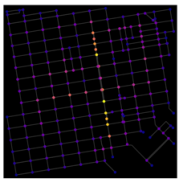
Geospatial

The following geospatial examples showcase different ways of performing network analyses using packages within the geospatial Python ecosystem. Example spatial files are stored directly in this directory. See the extended description for more details.

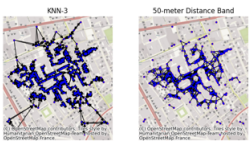
Delaunay graphs from geographic points



Graphs from a set of lines



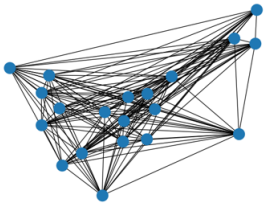
OpenStreetMap with OSMnx



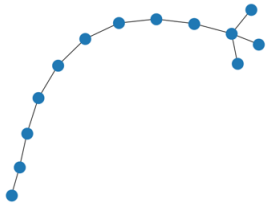
Graphs from geographic points

Graphs from Polygons

Subclass



Antigraph



Print Graph

Skip to main content

© Copyright 2004-2024, NetworkX Developers.

Created using Sphinx 7.2.6.

Built with the PyData Sphinx Theme 0.15.2.

Developer

Release: 3.3
Date: Apr 06, 2024

© Copyright 2004-2024, NetworkX Developers.

Created using Sphinx 7.2.6.

Built with the PyData Sphinx Theme 0.15.2.

Releases

We don’t use semantic versioning. The first number indicates that we have made a major API break (e.g., 1.x to 2.x), which has happened once and probably won’t happen again for some time. The point releases are new versions and may contain minor API breakage. Usually, this happens after a one cycle deprecation period.

Warning

Since we don’t normally make bug-fix only releases, it may not make sense for you to use `~=` as a pip version specifier.

Skip to main content

Skip to main content

Skip to main content

Skip to main content

Skip to main content

Skip to main content

© Copyright 2004-2024, NetworkX Developers.

Created using Sphinx 7.2.6.

Built with the PyData Sphinx Theme 0.15.2.

Software for Complex Networks

Release: 3.3
Date: Apr 06, 2024

NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. It provides:

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

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

Citing

To cite NetworkX please use the following publication:

Aric A. Hagberg, Daniel A. Schult and Pieter J. Swart, "Exploring network structure, dynamics, and function using NetworkX", in Proceedings of the 7th Python in Science Conference (SciPy2008), Gäel Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11–15, Aug 2008

PDF BibTeX

Audience

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

Python

Python is a powerful programming language that allows simple and flexible representations of networks as well as clear and concise expressions of network algorithms. Python has a vibrant and growing ecosystem of packages that NetworkX uses to provide more features such as numerical linear algebra and drawing. In order to make the most out of NetworkX you will want to know how to write basic programs in Python. Among the many guides to Python, we recommend the Python documentation and the text by Alex Martelli [Martelli03].

License

NetworkX is distributed with the 3-clause BSD license.


```
Copyright (C) 2004-2024, NetworkX Developers
Aric Hagberg <hagberg@lanl.gov>
Dan Schult <dschult@colgate.edu>
Pieter Swart <swart@lanl.gov>
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:

  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

  * Redistributions in binary form must reproduce the above
    copyright notice, this list of conditions and the following
    disclaimer in the documentation and/or other materials provided
    with the distribution.

  * Neither the name of the NetworkX Developers nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```


Bibliography

Skip to main content

[BA02] R. Albert and A.-L. Barabási, “Statistical mechanics of complex networks”, Reviews of Modern Physics, 74, pp. 47-97, 2002. https://arxiv.org/abs/cond-mat/0106096

[Bollobas01] B. Bollobás, “Random Graphs”, Second Edition, Cambridge University Press, 2001.

[BE05] U. Brandes and T. Erlebach, “Network Analysis: Methodological Foundations”, Lecture Notes in Computer Science, Volume 3418, Springer-Verlag, 2005.

[Diestel97] R. Diestel, “Graph Theory”, Springer-Verlag, 1997. http://diestel-graph-theory.com/index.html

[DM03] S.N. Dorogovtsev and J.F.F. Mendes, “Evolution of Networks”, Oxford University Press, 2003.

[Martelli03] A. Martelli, “Python in a Nutshell”, O’Reilly Media Inc, 2003.

[Newman03] M.E.J. Newman, “The Structure and Function of Complex Networks”, SIAM Review, 45, pp. 167-256, 2003. http://epubs.siam.org/doi/abs/10.1137/S003614450342480

[Sedgewick02] R. Sedgewick, “Algorithms in C: Parts 1-4: Fundamentals, Data Structure, Sorting, Searching”, Addison Wesley Professional, 3rd ed., 2002.

© Copyright 2004-2024, NetworkX Developers.

Built with the PyData Sphinx Theme 0.15.2.

Created using Sphinx 7.2.6.

Graph generators

Atlas

Generators for the small graph atlas.

`graph_atlas`(i)	Returns graph number `i` from the Graph Atlas.
`graph_atlas_g`()	Returns the list of all graphs with up to seven nodes named in the Graph Atlas.

Classic

Generators for some classic graphs.

The typical graph builder function is called as follows:

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

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

`balanced_tree`(r, h[, create_using])	Returns the perfectly balanced `r`-ary tree of height `h`.
`barbell_graph`(m1, m2[, create_using])	Returns the Barbell Graph: two complete graphs connected by a path.
`binomial_tree`(n[, create_using])	Returns the Binomial Tree of order n.
`complete_graph`(n[, create_using])	Return the complete graph `K_n` with n nodes.
`complete_multipartite_graph`(*subset_sizes)	Returns the complete multipartite graph with the specified subset sizes.
`circular_ladder_graph`(n[, create_using])	Returns the circular ladder graph CL_n of length n.

Skip to main content

	n nodes.
`cycle_graph`(n[, create_using])	Returns the cycle graph C_n of cyclically connected nodes.
`dorogovtsev_goltsev_mendes_graph`(n[, ...])	Returns the hierarchically constructed Dorogovtsev-Goltsev-Mendes graph.
`empty_graph`([n, create_using, default])	Returns the empty graph with n nodes and zero edges.
`full_rary_tree`(r, n[, create_using])	Creates a full r-ary tree of `n` nodes.
`kneser_graph`(n, k)	Returns the Kneser Graph with parameters `n` and `k`.
`ladder_graph`(n[, create_using])	Returns the Ladder graph of length n.
`lollipop_graph`(m, n[, create_using])	Returns the Lollipop Graph; `K_m` connected to `P_n`.
`null_graph`([create_using])	Returns the Null graph with no nodes or edges.
`path_graph`(n[, create_using])	Returns the Path graph `P_n` of linearly connected nodes.
`star_graph`(n[, create_using])	Return the star graph
`tadpole_graph`(m, n[, create_using])	Returns the (m,n)-tadpole graph; `C_m` connected to `P_n`.
`trivial_graph`([create_using])	Return the Trivial graph with one node (with label 0) and no edges.
`turan_graph`(n, r)	Return the Turan Graph
`wheel_graph`(n[, create_using])	Return the wheel graph

Expanders

Provides explicit constructions of expander graphs.

`margulis_gabber_galil_graph`(n[, create_using])	Returns the Margulis-Gabber-Galil undirected MultiGraph on `n^2` nodes.
`chordal_cycle_graph`(p[, create_using])	Returns the chordal cycle graph on `p` nodes.

Skip to main content

`paley_graph`(p[, create_using])	Returns the Paley $\frac{(p-1)}{2}$ -regular graph on p nodes.
`maybe_regular_expander`(n, d, *[, ...])	Utility for creating a random regular expander.
`is_regular_expander`(G, *[, epsilon])	Determines whether the graph G is a regular expander.
`random_regular_expander_graph`(n, d, *[, ...])	Returns a random regular expander graph on n nodes with degree d.

Lattice

Functions for generating grid graphs and lattices

The `grid_2d_graph()`, `triangular_lattice_graph()`, and `hexagonal_lattice_graph()` functions correspond to the three regular tilings of the plane, the square, triangular, and hexagonal tilings, respectively. `grid_graph()` and `hypercube_graph()` are similar for arbitrary dimensions. Useful relevant discussion can be found about Triangular Tiling, and Square, Hex and Triangle Grids

`grid_2d_graph`(m, n[, periodic, create_using])	Returns the two-dimensional grid graph.
`grid_graph`(dim[, periodic])	Returns the *n*-dimensional grid graph.
`hexagonal_lattice_graph`(m, n[, periodic, ...])	Returns an `m` by `n` hexagonal lattice graph.
`hypercube_graph`(n)	Returns the *n*-dimensional hypercube graph.
`triangular_lattice_graph`(m, n[, periodic, ...])	Returns the m by n triangular lattice graph.

Small

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

`LCF_graph`(n, shift_list, repeats[, create_using])	Return the cubic graph specified in LCF notation.
`bull_graph`([create_using])	Returns the Bull Graph
`chvatal_graph`([create_using])	Returns the Chvátal Graph
`cubical_graph`([create_using])	Returns the 3-regular Platonic Cubical Graph
`desargues_graph`([create_using])	Returns the Desargues Graph

Skip to main content

`diamond_graph`([create_using])	Returns the Diamond graph
`dodecahedral_graph`([create_using])	Returns the Platonic Dodecahedral graph.
`frucht_graph`([create_using])	Returns the Frucht Graph.
`heawood_graph`([create_using])	Returns the Heawood Graph, a (3,6) cage.
`hoffman_singleton_graph`()	Returns the Hoffman-Singleton Graph.
`house_graph`([create_using])	Returns the House graph (square with triangle on top)
`house_x_graph`([create_using])	Returns the House graph with a cross inside the house square.
`icosahedral_graph`([create_using])	Returns the Platonic Icosahedral graph.
`krackhardt_kite_graph`([create_using])	Returns the Krackhardt Kite Social Network.
`moebius_kantor_graph`([create_using])	Returns the Moebius-Kantor graph.
`octahedral_graph`([create_using])	Returns the Platonic Octahedral graph.
`pappus_graph`()	Returns the Pappus graph.
`petersen_graph`([create_using])	Returns the Petersen graph.
`sedgewick_maze_graph`([create_using])	Return a small maze with a cycle.
`tetrahedral_graph`([create_using])	Returns the 3-regular Platonic Tetrahedral graph.
`truncated_cube_graph`([create_using])	Returns the skeleton of the truncated cube.
`truncated_tetrahedron_graph`([create_using])	Returns the skeleton of the truncated Platonic tetrahedron.
`tutte_graph`([create_using])	Returns the Tutte graph.

Random Graphs

Generators for random graphs.

`fast_gnp_random_graph`(n, p[, seed, directed])	Returns a $G_{n,p}$ random graph, also known as an Erdős-Rényi graph or a binomial graph.
`gnp_random_graph`(n, p[, seed, directed])	Returns a $G_{n,p}$ random graph, also known as an

Skip to main content

`dense_gnm_random_graph`(n, m[, seed])	Returns a $G_{n,m}$ random graph.
`gnm_random_graph`(n, m[, seed, directed])	Returns a $G_{n,m}$ random graph.
`erdos_renyi_graph`(n, p[, seed, directed])	Returns a $G_{n,p}$ random graph, also known as an Erdős-Rényi graph or a binomial graph.
`binomial_graph`(n, p[, seed, directed])	Returns a $G_{n,p}$ random graph, also known as an Erdős-Rényi graph or a binomial graph.
`newman_watts_strogatz_graph`(n, k, p[, seed])	Returns a Newman–Watts–Strogatz small-world graph.
`watts_strogatz_graph`(n, k, p[, seed])	Returns a Watts–Strogatz small-world graph.
`connected_watts_strogatz_graph`(n, k, p[, ...])	Returns a connected Watts–Strogatz small-world graph.
`random_regular_graph`(d, n[, seed])	Returns a random d-regular graph on n nodes.
`barabasi_albert_graph`(n, m[, seed, ...])	Returns a random graph using Barabási–Albert preferential attachment
`dual_barabasi_albert_graph`(n, m1, m2, p[, ...])	Returns a random graph using dual Barabási–Albert preferential attachment
`extended_barabasi_albert_graph`(n, m, p, q[, ...])	Returns an extended Barabási–Albert model graph.
`powerlaw_cluster_graph`(n, m, p[, seed])	Holme and Kim algorithm for growing graphs with powerlaw degree distribution and approximate average clustering.
`random_kernel_graph`(n, kernel_integral[, ...])	Returns an random graph based on the specified kernel.
`random_lobster`(n, p1, p2[, seed])	Returns a random lobster graph.
`random_shell_graph`(constructor[, seed])	Returns a random shell graph for the constructor given.
`random_powerlaw_tree`(n[, gamma, seed, tries])	Returns a tree with a power law degree distribution.
`random_powerlaw_tree_sequence`(n[, gamma, ...])	Returns a degree sequence for a tree with a power law distribution.

Skip to main content

`random_kernel_graph`(n, kernel_integral[, ...])	Returns an random graph based on the specified kernel.

Duplication Divergence

Functions for generating graphs based on the “duplication” method.

These graph generators start with a small initial graph then duplicate nodes and (partially) duplicate their edges. These functions are generally inspired by biological networks.

`duplication_divergence_graph`(n, p[, seed])	Returns an undirected graph using the duplication-divergence model.
`partial_duplication_graph`(N, n, p, q[, seed])	Returns a random graph using the partial duplication model.

Degree Sequence

Generate graphs with a given degree sequence or expected degree sequence.

`configuration_model`(deg_sequence[, ...])	Returns a random graph with the given degree sequence.
`directed_configuration_model`(...[, ...])	Returns a directed_random graph with the given degree sequences.
`expected_degree_graph`(w[, seed, selfloops])	Returns a random graph with given expected degrees.
`havel_hakimi_graph`(deg_sequence[, create_using])	Returns a simple graph with given degree sequence constructed using the Havel-Hakimi algorithm.
`directed_havel_hakimi_graph`(in_deg_sequence, ...)	Returns a directed graph with the given degree sequences.
`degree_sequence_tree`(deg_sequence[, ...])	Make a tree for the given degree sequence.
`random_degree_sequence_graph`(sequence[, ...])	Returns a simple random graph with the given degree sequence.

Skip to main content

Random Clustered

Generate graphs with given degree and triangle sequence.

`random_clustered_graph`(joint_degree_sequence)	Generate a random graph with the given joint independent edge degree and triangle degree sequence.

Directed

Generators for some directed graphs, including growing network (GN) graphs and scale-free graphs.

`gn_graph`(n[, kernel, create_using, seed])	Returns the growing network (GN) digraph with `n` nodes.
`gnr_graph`(n, p[, create_using, seed])	Returns the growing network with redirection (GNR) digraph with `n` nodes and redirection probability `p`.
`gnc_graph`(n[, create_using, seed])	Returns the growing network with copying (GNC) digraph with `n` nodes.
`random_k_out_graph`(n, k, alpha[, ...])	Returns a random `k`-out graph with preferential attachment.
`scale_free_graph`(n[, alpha, beta, gamma, ...])	Returns a scale-free directed graph.

Geometric

Generators for geometric graphs.

`geometric_edges`(G, radius[, p, pos_name])	Returns edge list of node pairs within `radius` of each other.
`geographical_threshold_graph`(n, theta[, ...])	Returns a geographical threshold graph.
`navigable_small_world_graph`(n[, p, q, r, ...])	Returns a navigable small-world graph.
`random_geometric_graph`(n, radius[, dim, ...])	Returns a random geometric graph in the unit cube of dimensions `dim`.
`soft_random_geometric_graph`(n, radius[, ...])	Returns a soft random geometric graph in the unit cube.

Skip to main content

`thresholded_random_geometric_graph`(n, ...[, ...])	Returns a thresholded random geometric graph in the unit cube.
`waxman_graph`(n[, beta, alpha, L, domain, ...])	Returns a Waxman random graph.
`geometric_soft_configuration_graph`(*, beta)	Returns a random graph from the geometric soft configuration model.

Line Graph

Functions for generating line graphs.

`line_graph`(G[, create_using])	Returns the line graph of the graph or digraph `G`.
`inverse_line_graph`(G)	Returns the inverse line graph of graph G.

Ego Graph

Ego graph.

`ego_graph`(G, n[, radius, center, ...])	Returns induced subgraph of neighbors centered at node n within a given radius.

Stochastic

Functions for generating stochastic graphs from a given weighted directed graph.

`stochastic_graph`(G[, copy, weight])	Returns a right-stochastic representation of directed graph `G`.

AS graph

Generates graphs resembling the Internet Autonomous System network

`random_internet_as_graph`(n[, seed])	Generates a random undirected graph resembling the Internet AS network

Intersection

Skip to main content

Generators for random intersection graphs.

`uniform_random_intersection_graph`(n, m, p[, ...])	Returns a uniform random intersection graph.
`k_random_intersection_graph`(n, m, k[, seed])	Returns a intersection graph with randomly chosen attribute sets for each node that are of equal size (k).
`general_random_intersection_graph`(n, m, p[, ...])	Returns a random intersection graph with independent probabilities for connections between node and attribute sets.

Social Networks

Famous social networks.

`karate_club_graph`()	Returns Zachary's Karate Club graph.
`davis_southern_women_graph`()	Returns Davis Southern women social network.
`florentine_families_graph`()	Returns Florentine families graph.
`les_miserables_graph`()	Returns coappearance network of characters in the novel Les Miserables.

Community

Generators for classes of graphs used in studying social networks.

`caveman_graph`(l, k)	Returns a caveman graph of `l` cliques of size `k`.
`connected_caveman_graph`(l, k)	Returns a connected caveman graph of `l` cliques of size `k`.
`gaussian_random_partition_graph`(n, s, v, ...)	Generate a Gaussian random partition graph.
`LFR_benchmark_graph`(n, tau1, tau2, mu[, ...])	Returns the LFR benchmark graph.
`planted_partition_graph`(l, k, p_in, p_out[, ...])	Returns the planted l-partition graph.
`random_partition_graph`(sizes, p_in, p_out[, ...])	Returns the random partition graph with a partition of sizes.

Skip to main content

`ring_of_cliques`(num_cliques, clique_size)	Defines a "ring of cliques" graph.
`stochastic_block_model`(sizes, p[, nodelist, ...])	Returns a stochastic block model graph.
`windmill_graph`(n, k)	Generate a windmill graph.

Spectral

Generates graphs with a given eigenvector structure

`spectral_graph_forge`(G, alpha[, ...])	Returns a random simple graph with spectrum resembling that of `G`

Trees

Functions for generating trees.

The functions sampling trees at random in this module come in two variants: labeled and unlabeled. The labeled variants sample from every possible tree with the given number of nodes uniformly at random. The unlabeled variants sample from every possible *isomorphism class* of trees with the given number of nodes uniformly at random.

To understand the difference, consider the following example. There are two isomorphism classes of trees with four nodes. One is that of the path graph, the other is that of the star graph. The unlabeled variant will return a line graph or a star graph with probability 1/2.

The labeled variant will return the line graph with probability 3/4 and the star graph with probability 1/4, because there are more labeled variants of the line graph than of the star graph. More precisely, the line graph has an automorphism group of order 2, whereas the star graph has an automorphism group of order 6, so the line graph has three times as many labeled variants as the star graph, and thus three more chances to be drawn.

Additionally, some functions in this module can sample rooted trees and forests uniformly at random. A rooted tree is a tree with a designated root node. A rooted forest is a disjoint union of rooted trees.

`prefix_tree`(paths)	Creates a directed prefix tree from a list of paths.
`random_labeled_tree`(n, *[, seed])	Returns a labeled tree on `n` nodes chosen uniformly at random.
`random_labeled_rooted_tree`(n, *[, seed])	Returns a labeled rooted tree with `n` nodes.

Skip to main content

`random_labeled_rooted_forest`(n, *[, seed])	Returns a labeled rooted forest with `n` nodes.
`random_unlabeled_tree`(n, *[, ...])	Returns a tree or list of trees chosen randomly.
`random_unlabeled_rooted_tree`(n, *[, ...])	Returns a number of unlabeled rooted trees uniformly at random
`random_unlabeled_rooted_forest`(n, *[, q, ...])	Returns a forest or list of forests selected at random.

Non Isomorphic Trees

Implementation of the Wright, Richmond, Odlyzko and McKay (WROM) algorithm for the enumeration of all non-isomorphic free trees of a given order. Rooted trees are represented by level sequences, i.e., lists in which the i-th element specifies the distance of vertex i to the root.

`nonisomorphic_trees`(order[, create])	Generates lists of nonisomorphic trees
`number_of_nonisomorphic_trees`(order)	Returns the number of nonisomorphic trees

Triads

Functions that generate the triad graphs, that is, the possible digraphs on three nodes.

`triad_graph`(triad_name)	Returns the triad graph with the given name.

Joint Degree Sequence

Generate graphs with a given joint degree and directed joint degree

`is_valid_joint_degree`(joint_degrees)	Checks whether the given joint degree dictionary is realizable.
`joint_degree_graph`(joint_degrees[, seed])	Generates a random simple graph with the given joint degree dictionary.
`is_valid_directed_joint_degree`(in_degrees, ...)	Checks whether the given directed joint degree input is realizable
`directed_joint_degree_graph`(in_degrees, ...)	Generates a random simple directed graph with the joint degree.

Skip to main content

Mycielski

Functions related to the Mycielski Operation and the Mycielskian family of graphs.

`mycielskian`(G[, iterations])	Returns the Mycielskian of a simple, undirected graph G
`mycielski_graph`(n)	Generator for the n_th Mycielski Graph.

Harary Graph

Generators for Harary graphs

This module gives two generators for the Harary graph, which was introduced by the famous mathematician Frank Harary in his 1962 work [H]. The first generator gives the Harary graph that maximizes the node connectivity with given number of nodes and given number of edges. The second generator gives the Harary graph that minimizes the number of edges in the graph with given node connectivity and number of nodes.

References

[H] Harary, F. “The Maximum Connectivity of a Graph.” Proc. Nat. Acad. Sci. USA 48, 1142-1146, 1962.

`hnm_harary_graph`(n, m[, create_using])	Returns the Harary graph with given numbers of nodes and edges.
`hkn_harary_graph`(k, n[, create_using])	Returns the Harary graph with given node connectivity and node number.

Cographs

Generators for cographs

A cograph is a graph containing no path on four vertices. Cographs or P_4-free graphs can be obtained from a single vertex by disjoint union and complementation operations.

References

Skip to main content

[0] D.G. Corneil, H. Lerchs, L.Stewart Burlingham, “Complement reducible graphs”, Discrete Applied Mathematics, Volume 3, Issue 3, 1981, Pages 163-174, ISSN 0166-218X.

`random_cograph`(n[, seed])	Returns a random cograph with 2^n nodes.

Interval Graph

Generators for interval graph.

`interval_graph`(intervals)	Generates an interval graph for a list of intervals given.

Sudoku

Generator for Sudoku graphs

This module gives a generator for n-Sudoku graphs. It can be used to develop algorithms for solving or generating Sudoku puzzles.

A completed Sudoku grid is a 9x9 array of integers between 1 and 9, with no number appearing twice in the same row, column, or 3x3 box.

8 6 4 3 2 5 9 7 1	3 7 1 8 4 9 2 6 5	2 5 9 7 6 1 8 4 3
4 3 6 1 9 8 2 5 7	1 9 2 6 5 7 4 8 3	5 8 7 4 3 2 9 1 6
6 8 9 7 1 3 5 4 2	7 3 4 5 2 8 9 1 6	1 2 5 6 9 4 3 7 8

The Sudoku graph is an undirected graph with 81 vertices, corresponding to the cells of a Sudoku grid. It is a regular graph of degree 20. Two distinct vertices are adjacent if and only if the corresponding cells belong to the same row, column, or box. A completed Sudoku grid corresponds to a vertex coloring of the Sudoku graph with nine colors.

More generally, the n-Sudoku graph is a graph with n^4 vertices, corresponding to the cells of an n^2 by n^2 grid. Two distinct vertices are adjacent if and only if they belong to the same row, column, or n by n box.

Skip to main content

References

[1] Herzberg, A. M., & Murty, M. R. (2007). Sudoku squares and chromatic polynomials. Notices of the

© Copyright 2004-2024, NetworkX Developers.

Built with the PyData Sphinx Theme 0.15.2.

Created using Sphinx 7.2.6.

Reading and writing graphs

Skip to main content

© Copyright 2004-2024, NetworkX Developers.

Built with the PyData Sphinx Theme 0.15.2.

Created using Sphinx 7.2.6.

Algorithms

Skip to main content

Skip to main content

Skip to main content

Skip to main content

Skip to main content

Skip to main content

Skip to main content

Skip to main content

Skip to main content

Skip to main content

Skip to main content

© Copyright 2004-2024, NetworkX Developers.

Created using Sphinx 7.2.6.

Built with the PyData Sphinx Theme 0.15.2.

Drawing

NetworkX provides basic functionality for visualizing graphs, but its main goal is to enable graph analysis rather than perform graph visualization. In the future, graph visualization functionality may be removed from NetworkX or only available as an add-on package.

Proper graph visualization is hard, and we highly recommend that people visualize their graphs with tools dedicated to that task. Notable examples of dedicated and fully-featured graph visualization tools are Cytoscape, Gephi, Graphviz and, for LaTeX typesetting, PGF/TikZ. To use these and other such tools, you should export your NetworkX graph into a format that can be read by those tools. For example, Cytoscape can read the GraphML format, and so, `networkx.write_graphml(G, path)` might be an appropriate choice.

More information on the features provided here are available at

- matplotlib: http://matplotlib.org/
- pygraphviz: http://pygraphviz.github.io/

Matplotlib

Draw networks with matplotlib.

Examples

```
>>> G = nx.complete_graph(5)
>>> nx.draw(G)
```

See Also

- matplotlib
- `matplotlib.pyplot.scatter()`
- `matplotlib.patches.FancyArrowPatch`

`draw_networkx`(G[, pos, arrows, with_labels])	Draw the graph G using Matplotlib.
`draw_networkx_nodes`(G, pos[, nodelist, ...])	Draw the nodes of the graph G.
`draw_networkx_edges`(G, pos[, edgelist, ...])	Draw the edges of the graph G.
`draw_networkx_labels`(G, pos[, labels, ...])	Draw node labels on the graph G.
`draw_networkx_edge_labels`(G, pos[, ...])	Draw edge labels.
`draw_circular`(G, **kwargs)	Draw the graph `G` with a circular layout.
`draw_kamada_kawai`(G, **kwargs)	Draw the graph `G` with a Kamada-Kawai force-directed layout.
`draw_planar`(G, **kwargs)	Draw a planar networkx graph `G` with planar layout.
`draw_random`(G, **kwargs)	Draw the graph `G` with a random layout.
`draw_spectral`(G, **kwargs)	Draw the graph `G` with a spectral 2D layout.
`draw_spring`(G, **kwargs)	Draw the graph `G` with a spring layout.
`draw_shell`(G[, nlist])	Draw networkx graph `G` with shell layout.

Graphviz AGraph (dot)

Interface to pygraphviz AGraph class.

Examples

```
>>> G = nx.complete_graph(5)
>>> A = nx.nx_agraph.to_agraph(G)
>>> H = nx.nx_agraph.from_agraph(A)
```

See Also

- Pygraphviz: http://pygraphviz.github.io/
- Graphviz: https://www.graphviz.org
- DOT Language: http://www.graphviz.org/doc/info/lang.html

Skip to main content

`from_agraph`(A[, create_using])	Returns a NetworkX Graph or DiGraph from a PyGraphviz graph.
`to_agraph`(N)	Returns a pygraphviz graph from a NetworkX graph N.
`write_dot`(G, path)	Write NetworkX graph G to Graphviz dot format on path.
`read_dot`(path)	Returns a NetworkX graph from a dot file on path.
`graphviz_layout`(G[, prog, root, args])	Create node positions for G using Graphviz.
`pygraphviz_layout`(G[, prog, root, args])	Create node positions for G using Graphviz.

Graphviz with pydot

Import and export NetworkX graphs in Graphviz dot format using pydot.

Either this module or nx_agraph can be used to interface with graphviz.

Examples

```
>>> G = nx.complete_graph(5)
>>> PG = nx.nx_pydot.to_pydot(G)
>>> H = nx.nx_pydot.from_pydot(PG)
```

See Also

- pydot: erocarrera/pydot
- Graphviz: https://www.graphviz.org
- DOT Language: http://www.graphviz.org/doc/info/lang.html

`from_pydot`(P)	Returns a NetworkX graph from a Pydot graph.
`to_pydot`(N)	Returns a pydot graph from a NetworkX graph N.
`write_dot`(G, path)	Write NetworkX graph G to Graphviz dot format on path.
`read_dot`(path)	Returns a NetworkX `MultiGraph` or `MultiDiGraph` from the dot file with the passed path.

Skip to main content

`pydot_layout`(G[, prog, root])	Create node positions using `pydot` and Graphviz.

Graph Layout

Node positioning algorithms for graph drawing.

For `random_layout()` the possible resulting shape is a square of side [0, scale] (default: [0, 1]) Changing `center` shifts the layout by that amount.

For the other layout routines, the extent is [center - scale, center + scale] (default: [-1, 1]).

Warning: Most layout routines have only been tested in 2-dimensions.

`bipartite_layout`(G, nodes[, align, scale, ...])	Position nodes in two straight lines.
`bfs_layout`(G, start, *[, align, scale, center])	Position nodes according to breadth-first search algorithm.
`circular_layout`(G[, scale, center, dim])	Position nodes on a circle.
`kamada_kawai_layout`(G[, dist, pos, weight, ...])	Position nodes using Kamada-Kawai path-length cost-function.
`planar_layout`(G[, scale, center, dim])	Position nodes without edge intersections.
`random_layout`(G[, center, dim, seed])	Position nodes uniformly at random in the unit square.
`rescale_layout`(pos[, scale])	Returns scaled position array to (-scale, scale) in all axes.
`rescale_layout_dict`(pos[, scale])	Return a dictionary of scaled positions keyed by node
`shell_layout`(G[, nlist, rotate, scale, ...])	Position nodes in concentric circles.
`spring_layout`(G[, k, pos, fixed, ...])	Position nodes using Fruchterman-Reingold force-directed algorithm.
`spectral_layout`(G[, weight, scale, center, dim])	Position nodes using the eigenvectors of the graph Laplacian.
`spiral_layout`(G[, scale, center, dim, ...])	Position nodes in a spiral layout.
`multipartite_layout`(G[, subset_key, align, ...])	Position nodes in layers of straight lines.

Skip to main content

LaTeX Code

Export NetworkX graphs in LaTeX format using the TikZ library within TeX/LaTeX. Usually, you will want the drawing to appear in a figure environment so you use `to_latex(G, caption="A caption")`. If you want the raw drawing commands without a figure environment use **`to_latex_raw()`**. And if you want to write to a file instead of just returning the latex code as a string, use `write_latex(G, "filename.tex", caption="A caption")`.

To construct a figure with subfigures for each graph to be shown, provide `to_latex` or `write_latex` a list of graphs, a list of subcaptions, and a number of rows of subfigures inside the figure.

To be able to refer to the figures or subfigures in latex using `\\ref`, the keyword `latex_label` is available for figures and **`sub_labels`** for a list of labels, one for each subfigure.

We intend to eventually provide an interface to the TikZ Graph features which include e.g. layout algorithms.

Let us know via github what you'd like to see available, or better yet give us some code to do it, or even better make a github pull request to add the feature.

The TikZ approach

Drawing options can be stored on the graph as node/edge attributes, or can be provided as dicts keyed by node/edge to a string of the options for that node/edge. Similarly a label can be shown for each node/edge by specifying the labels as graph node/edge attributes or by providing a dict keyed by node/edge to the text to be written for that node/edge.

Options for the tikzpicture environment (e.g. "[scale=2]") can be provided via a keyword argument. Similarly default node and edge options can be provided through keywords arguments. The default node options are applied to the single TikZ "path" that draws all nodes (and no edges). The default edge options are applied to a TikZ "scope" which contains a path for each edge.

Examples

```
>>> G = nx.path_graph(3)
>>> nx.write_latex(G, "just_my_figure.tex", as_document=True)
>>> nx.write_latex(G, "my_figure.tex", caption="A path graph", latex_label="fig1")
>>> latex_code = nx.to_latex(G)  # a string rather than a file
```

You can change many features of the nodes and edges.

Skip to main content

```
>>> G = nx.path_graph(4, create_using=nx.DiGraph)
>>> pos = {n: (n, n) for n in G}  # nodes set on a line
```

```
>>> G.nodes[0]["style"] = "blue"
>>> G.nodes[2]["style"] = "line width=3,draw"
>>> G.nodes[3]["label"] = "Stop"
>>> G.edges[(0, 1)]["label"] = "1st Step"
>>> G.edges[(0, 1)]["label_opts"] = "near start"
>>> G.edges[(1, 2)]["style"] = "line width=3"
>>> G.edges[(1, 2)]["label"] = "2nd Step"
>>> G.edges[(2, 3)]["style"] = "green"
>>> G.edges[(2, 3)]["label"] = "3rd Step"
>>> G.edges[(2, 3)]["label_opts"] = "near end"
```

```
>>> nx.write_latex(G, "latex_graph.tex", pos=pos, as_document=True)
```

Then compile the LaTeX using something like `pdflatex latex_graph.tex` and view the pdf file created: `latex_graph.pdf`.

If you want **subfigures** each containing one graph, you can input a list of graphs.

```
>>> H1 = nx.path_graph(4)
>>> H2 = nx.complete_graph(4)
>>> H3 = nx.path_graph(8)
>>> H4 = nx.complete_graph(8)
>>> graphs = [H1, H2, H3, H4]
>>> caps = ["Path 4", "Complete graph 4", "Path 8", "Complete graph 8"]
>>> lbls = ["fig2a", "fig2b", "fig2c", "fig2d"]
>>> nx.write_latex(graphs, "subfigs.tex", n_rows=2, sub_captions=caps, sub_labels=lbls)
>>> latex_code = nx.to_latex(graphs, n_rows=2, sub_captions=caps, sub_labels=lbls)
```

```
>>> node_color = {0: "red", 1: "orange", 2: "blue", 3: "gray!90"}
>>> edge_width = {e: "line width=1.5" for e in H3.edges}
>>> pos = nx.circular_layout(H3)
>>> latex_code = nx.to_latex(H3, pos, node_options=node_color, edge_options=edge_width)
>>> print(latex_code)
\documentclass{report}
\usepackage{tikz}
\usepackage{subcaption}

\begin{document}
\begin{figure}
  \begin{tikzpicture}
      \draw
        (1.0, 0.0) node[red] (0){0}
        (0.707, 0.707) node[orange] (1){1}
        (-0.0, 1.0) node[blue] (2){2}
        (-0.707, 0.707) node[gray!90] (3){3}
        (-1.0, -0.0) node (4){4}
        (-0.707, -0.707) node (5){5}
        (0.0, -1.0) node (6){6}
        (0.707, -0.707) node (7){7};
```

Skip to main content

```
        \draw[line width=1.5] (1) to (2);
        \draw[line width=1.5] (2) to (3);
        \draw[line width=1.5] (3) to (4);
        \draw[line width=1.5] (4) to (5);
        \draw[line width=1.5] (5) to (6);
        \draw[line width=1.5] (6) to (7);
      \end{scope}
    \end{tikzpicture}
\end{figure}
\end{document}
```

Notes

If you want to change the preamble/postamble of the figure/document/subfigure environment, use the keyword arguments: `figure_wrapper`, `document_wrapper`, `subfigure_wrapper`. The default values are stored in private variables e.g. `nx.nx_layout._DOCUMENT_WRAPPER`

References

TikZ: https://tikz.dev/

TikZ options details: https://tikz.dev/tikz-actions

`to_latex_raw`(G[, pos, tikz_options, ...])	Return a string of the LaTeX/TikZ code to draw `G`
`to_latex`(Gbunch[, pos, tikz_options, ...])	Return latex code to draw the graph(s) in `Gbunch`

© Copyright 2004-2024, NetworkX Developers.

Created using Sphinx 7.2.6.

Built with the PyData Sphinx Theme 0.15.2.