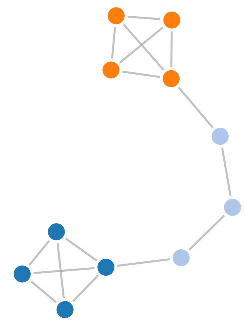
NetworkX

Network Analysis in Python

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



Software for complex networks

- Data structures for graphs, digraphs, and multigraphs
- Many standard graph algorithms
- Network structure and analysis measures
- Generators for classic graphs, random graphs, and synthetic networks
- Nodes can be "anything" (e.g., text, images, XML records)
- Edges can hold arbitrary data (e.g., weights, time-series)
- Open source 3-clause BSD license
- Well tested with over 90% code coverage
- Additional benefits from Python include fast prototyping, easy to teach, and multi-platform

Algorithms

A closer look at some of the algorithms and network analysis techniques provided by NetworkX.

Welcome to nx-guides!

Contents

launch binder

This site provides educational materials officially developed and curated by the NetworkX community. The goal of the repository is to provide high-quality educational resources for learning about network analysis and graph theory with NetworkX. Examples include:

- Long-form narrative documentation, such as tutorials
- In-depth examinations of common graph and network algorithms and their implementations in NetworkX
- Demonstrations or domain-specific applications of NetworkX highlighting best-practices for network analysis.

About

The educational materials are in the form of markdown-based Jupyter notebooks, so everything is interactive! You can follow along yourself:

1. *on binder*, by clicking on the launch button at the top of this page, or the rocket icon in the upper-right corner of any

of the pages, or

2. *locally*, by cloning the repository (see the octocat icon above) and running `jupyter notebook`.

Contents

Graph Generators

A closer look at the functions provided by NetworkX to create interesting graphs.

Algorithms

- constraint
- effective_size
- local_constraint

- Summarization
  - dedensify
  - snap_aggregation
- Swap
  - double_edge_swap
  - directed_edge_swap
  - connected_double_edge_swap
- Threshold Graphs
  - find_threshold_graph
  - is_threshold_graph
- Tournament
  - hamiltonian_path
  - is_reachable
  - is_strongly_connected
  - is_tournament
  - random_tournament
  - score_sequence
- Traversal
  - Depth First Search
  - Breadth First Search

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)
```

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)]
```

```
>>> assert list(DG.edges) == [(2, 1), (2, 4), (1, 3), (1, 2)]
```

Examining elements of a graph

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})
```

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)
```

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
>>> list(H.edges())
[(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.

```
>>> 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

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')
>>> G.add_edges_from([(1, 2, {'color': 'blue'}), (2, 3, {'weight': 8})])
>>> 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 `degree` reports the sum of `in_degree` and `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
```

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.
`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-Rényi graph or a binomial graph.
`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]
>>> 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.

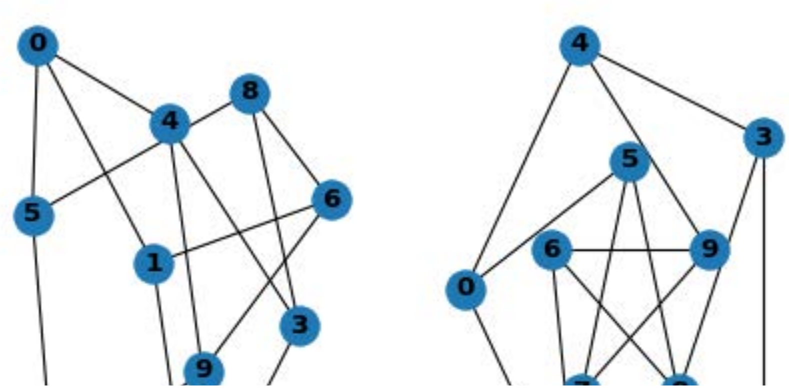
First import Matplotlib's plot interface (pylab works too)

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

To test if the import of `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')
```

(png, hires.png, pdf)



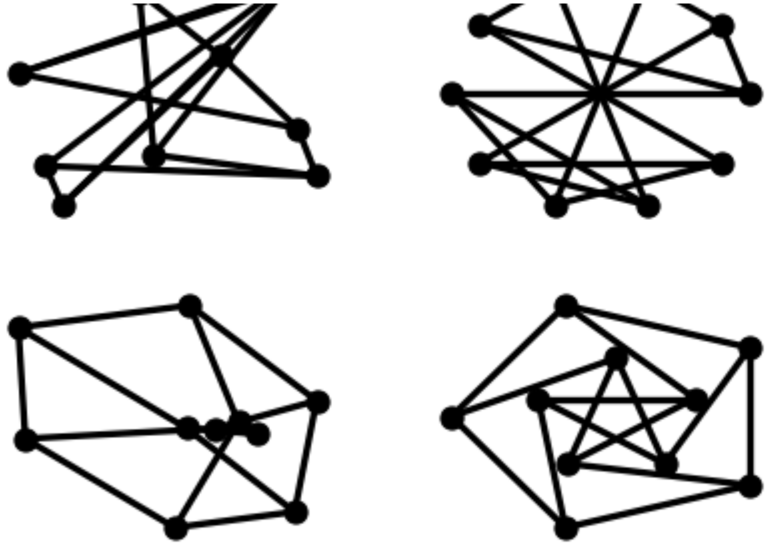
when drawing to an interactive display. Note that you may need to issue a Matplotlib

```
>>> 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)
```

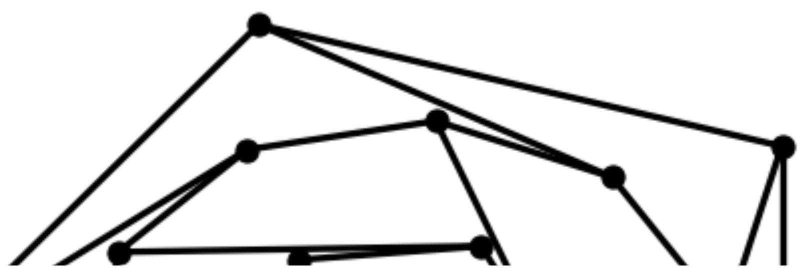
(png, hires.png, pdf)



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)
```

(png, hires.png, pdf)



To save drawings to a file, use, for example

```
>>> nx.draw(G)
>>> plt.savefig("path.png")
```

This function writes to the file `path.png` in the local directory. If Graphviz and PyGraphviz or pydot, are available on your system, you can also use `networkx.drawing.nx_agraph.graphviz_layout` or `networkx.drawing.nx_pydot.graphviz_layout` to get the node positions, or write the graph in dot format for further processing.

```
>>> from networkx.drawing.nx_pydot import write_dot
>>> pos = nx.nx_agraph.graphviz_layout(G)
>>> nx.draw(G, pos=pos)
>>> write_dot(G, 'file.dot')
```

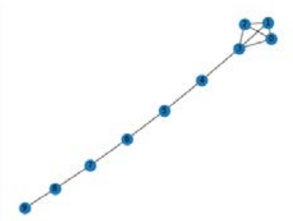
See Drawing for additional details.

- Download this page as a Python code file;
- Download this page as a Jupyter notebook (no outputs);
- Download this page as a Jupyter notebook (with outputs).

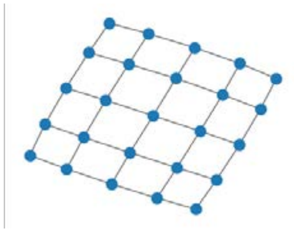
Gallery

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

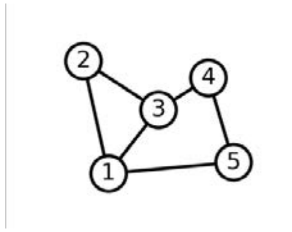
Basic



Properties

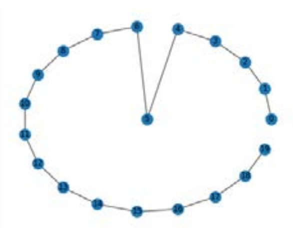


Read and write graphs.



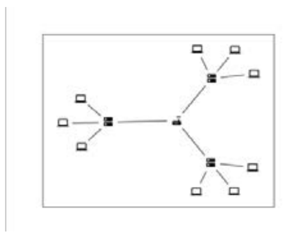
Simple graph

Drawing

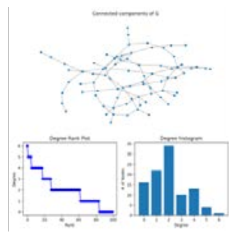


Custom Node

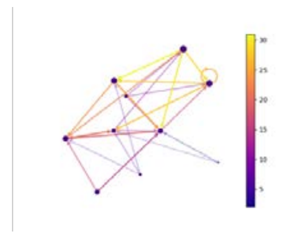
Chess Masters



Custom node icons



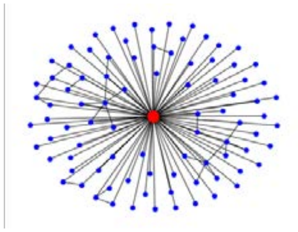
Degree Analysis



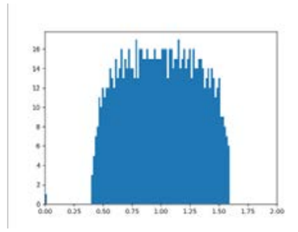
Directed Graph

Position

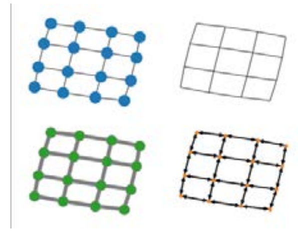
Edge Colormap



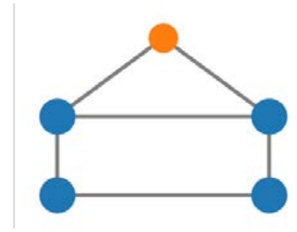
Ego Graph



Eigenvalues

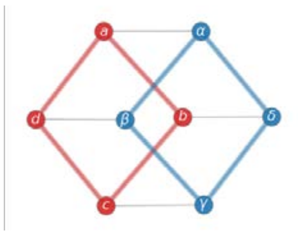


Four Grids

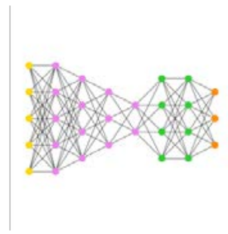


House With Colors

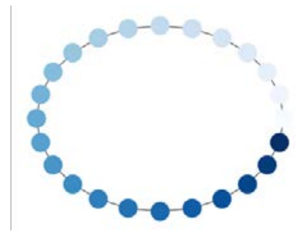
Knuth Miles



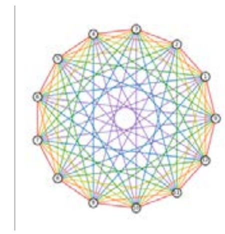
Labels And Colors



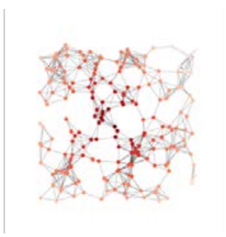
Multipartite Layout



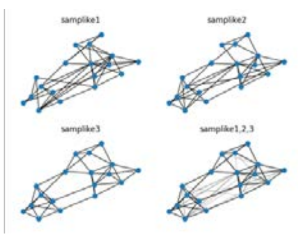
Node Colormap



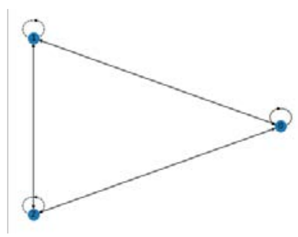
Rainbow Coloring



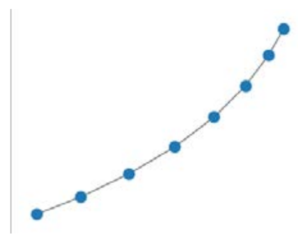
Random Geometric Graph



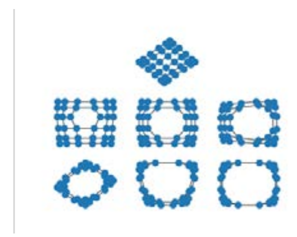
Sampson



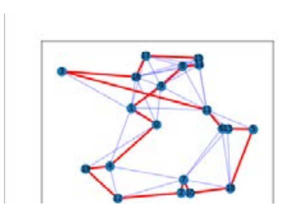
Self-loops

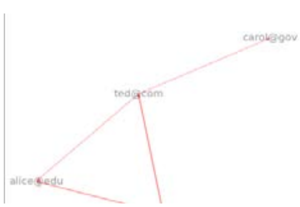


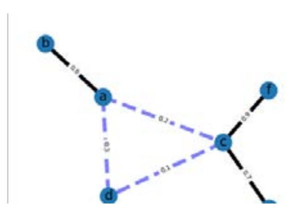
Simple Path



Spectral Embedding







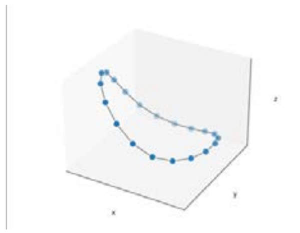
Traveling Salesman Problem

Unix Email

Weighted Graph

3D Drawing

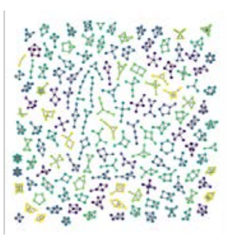
Mayavi2



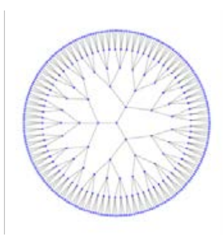
Basic matplotlib

Graphviz Layout

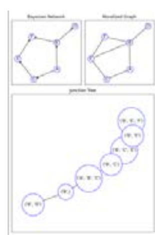
Examples using Graphviz layouts with `nx_pylab` for drawing. These examples need Graphviz and PyGraphviz.



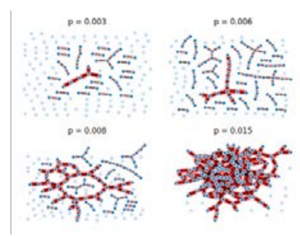
Atlas



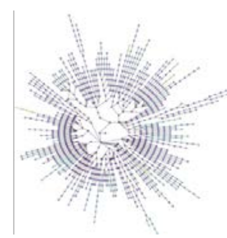
Circular Tree



Decomposition



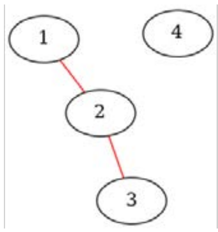
Giant Component



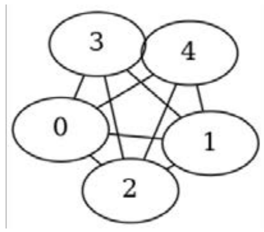
Lanl Routes

Graphviz Drawing

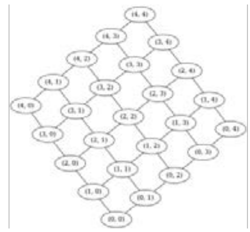
Examples using Graphviz for layout and drawing via `nx_agraph`. These examples need Graphviz and PyGraphviz.



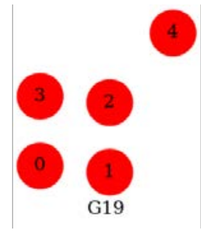
Attributes



Conversion

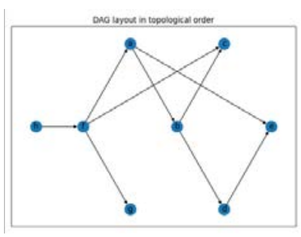


2D Grid

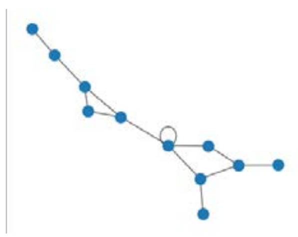


Atlas

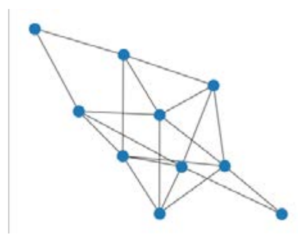
Graph



DAG - Topological Layout



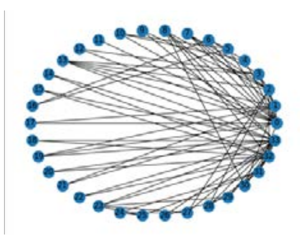
Degree Sequence



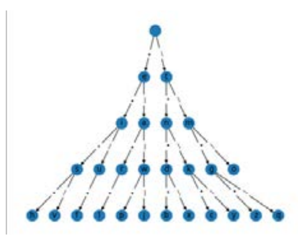
Erdos Renyi

Expected Degree Sequence

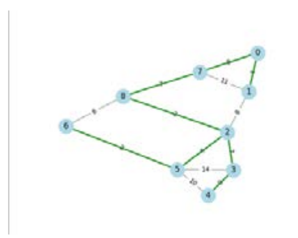
Football



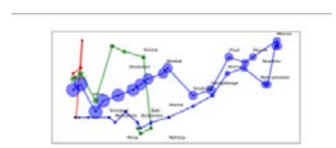
Karate Club



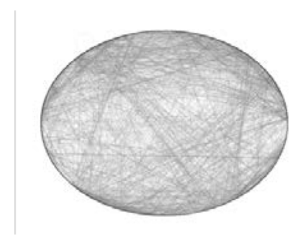
Morse Trie



Minimum Spanning



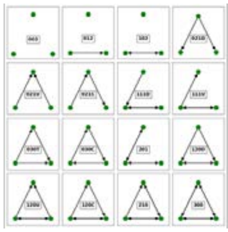
Napoleon Russian



Roget

Tree

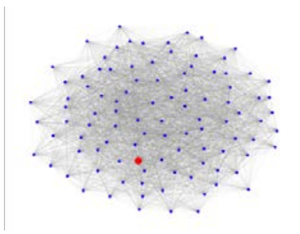
Campaign



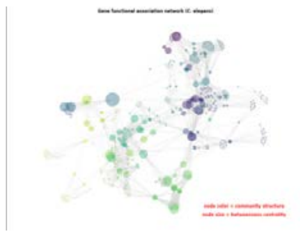
Triads

Words/Ladder Graph

Algorithms

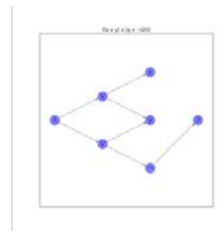


Beam Search



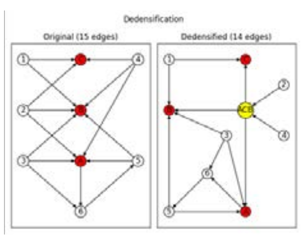
Betweenness Centrality

Blockmodel

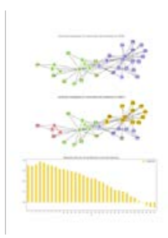


Circuits

Davis Club

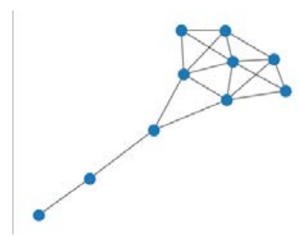


Dedensification

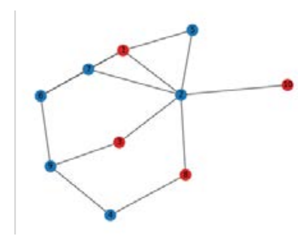


Community Detection using Girvan-Newman

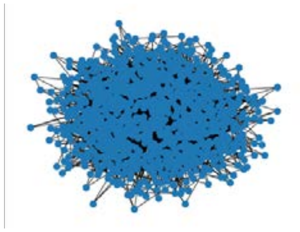
Iterated Dynamical Systems



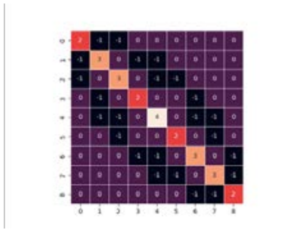
Krackhardt Centrality



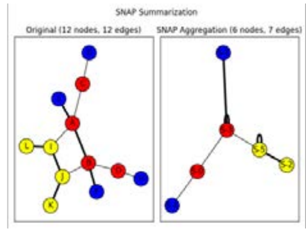
Maximum Independent Set



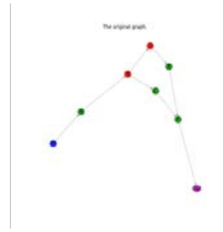
Parallel Betweenness



Reverse Cuthill--McKee



SNAP Graph Summary

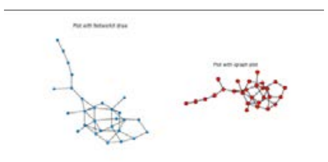


Subgraphs

External libraries

Examples of using NetworkX with external libraries.

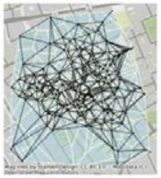
Javascript



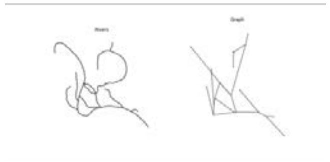
igraph

Geospatial

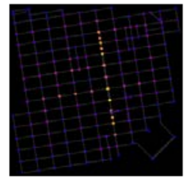
The following geospatial examples showcase different ways of performing network analyses using packages within the geospatial Python ecosystem. Example spatial files are stored directly in this directory. See the extended description for more details.



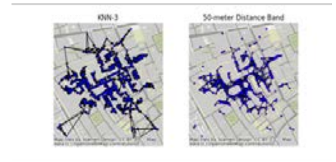
Delaunay graphs from geographic points



Graphs from a set of lines



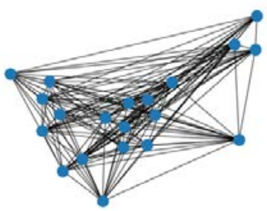
OpenStreetMap with OSMnx



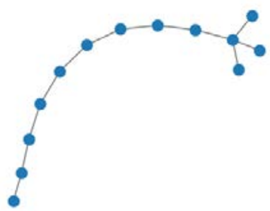
Graphs from geographic points

Graphs from Polygons

Subclass



Antigraph



Print Graph

Download all examples in Python source code: auto_examples_python.zip

Download all examples in Jupyter notebooks: auto_examples_jupyter.zip

Gallery generated by Sphinx-Gallery

Drawing

NetworkX provides basic functionality for visualizing graphs, but its main goal is to enable graph analysis rather than perform grap 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 b 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

Skip to main content

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

`draw`(G[, pos, ax])	Draw the graph G with Matplotlib.
`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

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

`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.

Skip to main content

`graphviz_layout`(G[, prog, root])	Create node positions using Pydot and Graphviz.
`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 b 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.
`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.

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")`.