To copy node attributes from the original graph to the new graph, you can use a dictionary like the one constructed in the above example:

```
>>> for source, nodes in sources.items():
...     for v in nodes:
...         B.nodes[v].update(G.nodes[source])
```

Notes

This function is not idempotent in the sense that the node labels in the returned branching may be uniquely generated each time the function is invoked. In fact, the node labels may not be integers; in order to relabel the nodes to be more readable, you can use the `networkx.convert_node_labels_to_integers()` function.

The current implementation of this function uses `networkx.prefix_tree()`, so it is subject to the limitations of that function.

3.22 Distance Measures

Graph diameter, radius, eccentricity and other properties.

barycenter(G[, weight, attr, sp])	Calculate barycenter of a connected graph, optionally with edge weights.
center(G[, e, usebounds])	Returns the center of the graph G.
diameter(G[, e, usebounds])	Returns the diameter of the graph G.
eccentricity(G[, v, sp])	Returns the eccentricity of nodes in G.
extrema_bounding(G[, compute])	Compute requested extreme distance metric of undirected graph G
periphery(G[, e, usebounds])	Returns the periphery of the graph G.
radius(G[, e, usebounds])	Returns the radius of the graph G.
resistance_distance(G, nodeA, nodeB[, ...])	Returns the resistance distance between node A and node B on graph G.

3.22.1 networkx.algorithms.distance_measures.barycenter

barycenter(*G*, *weight=None*, *attr=None*, *sp=None*)

Calculate barycenter of a connected graph, optionally with edge weights.

The *barycenter* a *connected* graph G is the subgraph induced by the set of its nodes v minimizing the objective function

$$\sum_{u \in V(G)} d_G(u, v),$$

where d_G is the (possibly weighted) *path length*. The barycenter is also called the *median*. See [?], p. 78.

Parameters

- **G** (*networkx.Graph*) – The connected graph G.
- **weight** (str, optional) – Passed through to *shortest_path_length()*.
- **attr** (str, optional) – If given, write the value of the objective function to each node's `attr` attribute. Otherwise do not store the value.

- **sp** (*dict of dicts, optional*) – All pairs shortest path lengths as a dictionary of dictionaries

Returns Nodes of `G` that induce the barycenter of `G`.

Return type `list`

Raises

- ***networkx.NetworkXNoPath*** – If `G` is disconnected. `G` may appear disconnected to *barycenter()* if `sp` is given but is missing shortest path lengths for any pairs.
- **ValueError** – If `sp` and `weight` are both given.

See also:

center(), *periphery()*

3.22.2 networkx.algorithms.distance_measures.center

center(*G*, *e=None*, *usebounds=False*)

Returns the center of the graph G.

The center is the set of nodes with eccentricity equal to radius.

Parameters

- **G** (*NetworkX graph*) – A graph
- **e** (*eccentricity dictionary, optional*) – A precomputed dictionary of eccentricities.

Returns **c** – List of nodes in center

Return type list

See also:

barycenter(), *periphery()*

3.22.3 networkx.algorithms.distance_measures.diameter

diameter(*G*, *e=None*, *usebounds=False*)

Returns the diameter of the graph G.

The diameter is the maximum eccentricity.

Parameters

- **G** (*NetworkX graph*) – A graph
- **e** (*eccentricity dictionary, optional*) – A precomputed dictionary of eccentricities.

Returns **d** – Diameter of graph

Return type integer

See also:

eccentricity()

3.22.4 networkx.algorithms.distance_measures.eccentricity

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

Returns the eccentricity of nodes in G.

The eccentricity of a node v is the maximum distance from v to all other nodes in G.

Parameters

- **G** (*NetworkX graph*) – A graph
- **v** (*node, optional*) – Return value of specified node
- **sp** (*dict of dicts, optional*) – All pairs shortest path lengths as a dictionary of dictionaries

Returns **ecc** – A dictionary of eccentricity values keyed by node.

Return type dictionary

3.22.5 networkx.algorithms.distance_measures.extrema_bounding

extrema_bounding(*G*, *compute='diameter'*)

Compute requested extreme distance metric of undirected graph G

Computation is based on smart lower and upper bounds, and in practice linear in the number of nodes, rather than quadratic (except for some border cases such as complete graphs or circle shaped graphs).

Parameters

- **G** (*NetworkX graph*) – An undirected graph
- **compute** (*string denoting the requesting metric*) – "diameter" for the maximal eccentricity value, "radius" for the minimal eccentricity value, "periphery" for the set of nodes with eccentricity equal to the diameter "center" for the set of nodes with eccentricity equal to the radius

Returns **value** – int for "diameter" and "radius" or list of nodes for "center" and "periphery"

Return type value of the requested metric

Raises ***NetworkXError*** – If the graph consists of multiple components

Notes

This algorithm was proposed in the following papers:

F.W. Takes and W.A. Kosters, Determining the Diameter of Small World Networks, in Proceedings of the 20th ACM International Conference on Information and Knowledge Management (CIKM 2011), pp. 1191-1196, 2011. doi: https://doi.org/10.1145/2063576.2063748

F.W. Takes and W.A. Kosters, Computing the Eccentricity Distribution of Large Graphs, Algorithms 6(1): 100-118, 2013. doi: https://doi.org/10.3390/a6010100

M. Borassi, P. Crescenzi, M. Habib, W.A. Kosters, A. Marino and F.W. Takes, Fast Graph Diameter and Radius BFS-Based Computation in (Weakly Connected) Real-World Graphs, Theoretical Computer Science 586: 59-80, 2015. doi: https://doi.org/10.1016/j.tcs.2015.02.033

3.22.6 networkx.algorithms.distance_measures.periphery

periphery(*G*, *e=None*, *usebounds=False*)
Returns the periphery of the graph G.

The periphery is the set of nodes with eccentricity equal to the diameter.

Parameters

- **G** (*NetworkX graph*) – A graph
- **e** (*eccentricity dictionary, optional*) – A precomputed dictionary of eccentricities.

Returns **p** – List of nodes in periphery

Return type list

See also:

barycenter(), *center()*

3.22.7 networkx.algorithms.distance_measures.radius

radius(*G*, *e=None*, *usebounds=False*)
Returns the radius of the graph G.

The radius is the minimum eccentricity.

Parameters

- **G** (*NetworkX graph*) – A graph
- **e** (*eccentricity dictionary, optional*) – A precomputed dictionary of eccentricities.

Returns **r** – Radius of graph

Return type integer

3.22.8 networkx.algorithms.distance_measures.resistance_distance

resistance_distance(*G*, *nodeA*, *nodeB*, *weight=None*, *invert_weight=True*)
Returns the resistance distance between node A and node B on graph G.

The resistance distance between two nodes of a graph is akin to treating the graph as a grid of resistorses with a resistance equal to the provided weight.

If weight is not provided, then a weight of 1 is used for all edges.

Parameters

- **G** (*NetworkX graph*) – A graph
- **nodeA** (*node*) – A node within graph G.
- **nodeB** (*node*) – A node within graph G, exclusive of Node A.
- **weight** (*string or None, optional (default=None)*) – The edge data key used to compute the resistance distance. If None, then each edge has weight 1.
- **invert_weight** (*boolean (default=True)*) – Proper calculation of resistance distance requires building the Laplacian matrix with the reciprocal of the weight. Not required if the weight is already inverted. Weight cannot be zero.

Returns **rd** – Value of effective resistance distance

Return type float

Notes

Overview discussion: * https://en.wikipedia.org/wiki/Resistance_distance * http://mathworld.wolfram.com/ResistanceDistance.html

Additional details: Vaya Sapobi Samui Vos, "Methods for determining the effective resistance," M.S., Mathematisch Instituut, Universiteit Leiden, Leiden, Netherlands, 2016 Available: Link to thesis

3.23 Distance-Regular Graphs

3.23.1 Distance-regular graphs

is_distance_regular(G)	Returns True if the graph is distance regular, False otherwise.
is_strongly_regular(G)	Returns True if and only if the given graph is strongly regular.
intersection_array(G)	Returns the intersection array of a distance-regular graph.
global_parameters(b, c)	Returns global parameters for a given intersection array.

3.23.2 networkx.algorithms.distance_regular.is_distance_regular

is_distance_regular(*G*)

Returns True if the graph is distance regular, False otherwise.

A connected graph G is distance-regular if for any nodes x,y and any integers i,j=0,1,. . . ,d (where d is the graph diameter), the number of vertices at distance i from x and distance j from y depends only on i,j and the graph distance between x and y, independently of the choice of x and y.

Parameters **G** (*Networkx graph (undirected)*)

Returns True if the graph is Distance Regular, False otherwise

Return type bool

Examples

```
>>> G=nx.hypercube_graph(6)
>>> nx.is_distance_regular(G)
True
```

See also:

intersection_array(), *global_parameters()*

Notes

For undirected and simple graphs only

- **Warning** (If `source` provided is not the start node of an Euler path)
- *will raise error even if an Euler Path exists.*

3.28 Flows

3.28.1 Maximum Flow

maximum_flow(flowG, _s, _t[, capacity, ...])	Find a maximum single-commodity flow.
maximum_flow_value(flowG, _s, _t[, ...])	Find the value of maximum single-commodity flow.
minimum_cut(flowG, _s, _t[, capacity, flow_func])	Compute the value and the node partition of a minimum (s, t)-cut.
minimum_cut_value(flowG, _s, _t[, capacity, ...])	Compute the value of a minimum (s, t)-cut.

networkx.algorithms.flow.maximum_flow

maximum_flow (*flowG*, *_s*, *_t*, *capacity='capacity'*, *flow_func=None*, ***kwargs*)

Find a maximum single-commodity flow.

Parameters

- **flowG** (*NetworkX graph*) – Edges of the graph are expected to have an attribute called 'capacity'. If this attribute is not present, the edge is considered to have infinite capacity.
- **_s** (*node*) – Source node for the flow.
- **_t** (*node*) – Sink node for the flow.
- **capacity** (*string*) – Edges of the graph G are expected to have an attribute capacity that indicates how much flow the edge can support. If this attribute is not present, the edge is considered to have infinite capacity. Default value: 'capacity'.
- **flow_func** (*function*) – A function for computing the maximum flow among a pair of nodes in a capacitated graph. The function has to accept at least three parameters: a Graph or Digraph, a source node, and a target node. And return a residual network that follows NetworkX conventions (see Notes). If flow_func is None, the default maximum flow function (*preflow_push()*) is used. See below for alternative algorithms. The choice of the default function may change from version to version and should not be relied on. Default value: None.
- **kwargs** (*Any other keyword parameter is passed to the function that*) – computes the maximum flow.

Returns

- **flow_value** (*integer, float*) – Value of the maximum flow, i.e., net outflow from the source.
- **flow_dict** (*dict*) – A dictionary containing the value of the flow that went through each edge.

Raises

- ***NetworkXError*** – The algorithm does not support MultiGraph and MultiDiGraph. If the input graph is an instance of one of these two classes, a NetworkXError is raised.
- ***NetworkXUnbounded*** – If the graph has a path of infinite capacity, the value of a feasible flow on the graph is unbounded above and the function raises a NetworkXUnbounded.

See also:

maximum_flow_value(), *minimum_cut()*, *minimum_cut_value()*, *edmonds_karp()*, *preflow_push()*, *shortest_augmenting_path()*

Notes

The function used in the flow_func parameter has to return a residual network that follows NetworkX conventions:

The residual network `R` from an input graph `G` has the same nodes as `G`. `R` is a DiGraph that contains a pair of edges `(u, v)` and `(v, u)` iff `(u, v)` is not a self-loop, and at least one of `(u, v)` and `(v, u)` exists in `G`.

For each edge `(u, v)` in `R`, `R[u][v]['capacity']` is equal to the capacity of `(u, v)` in `G` if it exists in `G` or zero otherwise. If the capacity is infinite, `R[u][v]['capacity']` will have a high arbitrary finite value that does not affect the solution of the problem. This value is stored in `R.graph['inf']`. For each edge `(u, v)` in `R`, `R[u][v]['flow']` represents the flow function of `(u, v)` and satisfies `R[u][v]['flow'] == -R[v][u]['flow']`.

The flow value, defined as the total flow into `t`, the sink, is stored in `R.graph['flow_value']`. Reachability to `t` using only edges `(u, v)` such that `R[u][v]['flow'] < R[u][v]['capacity']` induces a minimum `s-t` cut.

Specific algorithms may store extra data in `R`.

The function should supports an optional boolean parameter value_only. When True, it can optionally terminate the algorithm as soon as the maximum flow value and the minimum cut can be determined.

Examples

```
>>> import networkx as nx
>>> G = nx.DiGraph()
>>> G.add_edge('x','a', capacity=3.0)
>>> G.add_edge('x','b', capacity=1.0)
>>> G.add_edge('a','c', capacity=3.0)
>>> G.add_edge('b','c', capacity=5.0)
>>> G.add_edge('b','d', capacity=4.0)
>>> G.add_edge('d','e', capacity=2.0)
>>> G.add_edge('c','y', capacity=2.0)
>>> G.add_edge('e','y', capacity=3.0)
```

maximum_flow returns both the value of the maximum flow and a dictionary with all flows.

```
>>> flow_value, flow_dict = nx.maximum_flow(G, 'x', 'y')
>>> flow_value
3.0
>>> print(flow_dict['x']['b'])
1.0
```

You can also use alternative algorithms for computing the maximum flow by using the flow_func parameter.

```
>>> from networkx.algorithms.flow import shortest_augmenting_path
>>> flow_value == nx.maximum_flow(G, 'x', 'y',
...                               flow_func=shortest_augmenting_path)[0]
True
```

networkx.algorithms.flow.maximum_flow_value

maximum_flow_value(*flowG*, *_s*, *_t*, *capacity='capacity'*, *flow_func=None*, ***kwargs*)

Find the value of maximum single-commodity flow.

Parameters

- **flowG** (*NetworkX graph*) – Edges of the graph are expected to have an attribute called 'capacity'. If this attribute is not present, the edge is considered to have infinite capacity.
- **_s** (*node*) – Source node for the flow.
- **_t** (*node*) – Sink node for the flow.
- **capacity** (*string*) – Edges of the graph G are expected to have an attribute capacity that indicates how much flow the edge can support. If this attribute is not present, the edge is considered to have infinite capacity. Default value: 'capacity'.
- **flow_func** (*function*) – A function for computing the maximum flow among a pair of nodes in a capacitated graph. The function has to accept at least three parameters: a Graph or Digraph, a source node, and a target node. And return a residual network that follows NetworkX conventions (see Notes). If flow_func is None, the default maximum flow function (*preflow_push()*) is used. See below for alternative algorithms. The choice of the default function may change from version to version and should not be relied on. Default value: None.
- **kwargs** (*Any other keyword parameter is passed to the function that*) – computes the maximum flow.

Returns **flow_value** – Value of the maximum flow, i.e., net outflow from the source.

Return type integer, float

Raises

- ***NetworkXError*** – The algorithm does not support MultiGraph and MultiDiGraph. If the input graph is an instance of one of these two classes, a NetworkXError is raised.
- ***NetworkXUnbounded*** – If the graph has a path of infinite capacity, the value of a feasible flow on the graph is unbounded above and the function raises a NetworkXUnbounded.

See also:

maximum_flow(), *minimum_cut()*, *minimum_cut_value()*, *edmonds_karp()*, *preflow_push()*, *shortest_augmenting_path()*

Notes

The function used in the flow_func parameter has to return a residual network that follows NetworkX conventions:

The residual network `R` from an input graph `G` has the same nodes as `G`. `R` is a DiGraph that contains a pair of edges `(u, v)` and `(v, u)` iff `(u, v)` is not a self-loop, and at least one of `(u, v)` and `(v, u)` exists in `G`.

For each edge `(u, v)` in `R`, `R[u][v]['capacity']` is equal to the capacity of `(u, v)` in `G` if it exists in `G` or zero otherwise. If the capacity is infinite, `R[u][v]['capacity']` will have a high arbitrary finite value that does not affect the solution of the problem. This value is stored in `R.graph['inf']`. For each edge `(u, v)` in `R`, `R[u][v]['flow']` represents the flow function of `(u, v)` and satisfies `R[u][v]['flow'] == -R[v][u]['flow']`.

The flow value, defined as the total flow into `t`, the sink, is stored in `R.graph['flow_value']`. Reachability to `t` using only edges `(u, v)` such that `R[u][v]['flow'] < R[u][v]['capacity']` induces a minimum `s-t` cut.

Specific algorithms may store extra data in `R`.

The function should supports an optional boolean parameter value_only. When True, it can optionally terminate the algorithm as soon as the maximum flow value and the minimum cut can be determined.

Examples

```
>>> import networkx as nx
>>> G = nx.DiGraph()
>>> G.add_edge('x','a', capacity=3.0)
>>> G.add_edge('x','b', capacity=1.0)
>>> G.add_edge('a','c', capacity=3.0)
>>> G.add_edge('b','c', capacity=5.0)
>>> G.add_edge('b','d', capacity=4.0)
>>> G.add_edge('d','e', capacity=2.0)
>>> G.add_edge('c','y', capacity=2.0)
>>> G.add_edge('e','y', capacity=3.0)
```

maximum_flow_value computes only the value of the maximum flow:

```
>>> flow_value = nx.maximum_flow_value(G, 'x', 'y')
>>> flow_value
3.0
```

You can also use alternative algorithms for computing the maximum flow by using the flow_func parameter.

```
>>> from networkx.algorithms.flow import shortest_augmenting_path
>>> flow_value == nx.maximum_flow_value(G, 'x', 'y',
...                                     flow_func=shortest_augmenting_path)
True
```

networkx.algorithms.flow.minimum_cut

minimum_cut (*flowG*, *_s*, *_t*, *capacity='capacity'*, *flow_func=None*, ***kwargs*)

Compute the value and the node partition of a minimum (s, t)-cut.

Use the max-flow min-cut theorem, i.e., the capacity of a minimum capacity cut is equal to the flow value of a maximum flow.

Parameters

- **flowG** (*NetworkX graph*) – Edges of the graph are expected to have an attribute called 'capacity'. If this attribute is not present, the edge is considered to have infinite capacity.
- **_s** (*node*) – Source node for the flow.
- **_t** (*node*) – Sink node for the flow.
- **capacity** (*string*) – Edges of the graph G are expected to have an attribute capacity that indicates how much flow the edge can support. If this attribute is not present, the edge is considered to have infinite capacity. Default value: 'capacity'.
- **flow_func** (*function*) – A function for computing the maximum flow among a pair of nodes in a capacitated graph. The function has to accept at least three parameters: a Graph or

`v)` in `R`, `R[u][v]['flow']` represents the flow function of `(u, v)` and satisfies `R[u][v]['flow'] == -R[v][u]['flow']`.

The flow value, defined as the total flow into `t`, the sink, is stored in `R.graph['flow_value']`. If `cutoff` is not specified, reachability to `t` using only edges `(u, v)` such that `R[u][v]['flow'] < R[u][v]['capacity']` induces a minimum `s-t` cut.

3.28.9 Network Simplex

network_simplex(G[, demand, capacity, weight])	Find a minimum cost flow satisfying all demands in digraph G.
min_cost_flow_cost(G[, demand, capacity, weight])	Find the cost of a minimum cost flow satisfying all demands in digraph G.
min_cost_flow(G[, demand, capacity, weight])	Returns a minimum cost flow satisfying all demands in digraph G.
cost_of_flow(G, flowDict[, weight])	Compute the cost of the flow given by flowDict on graph G.
max_flow_min_cost(G, s, t[, capacity, weight])	Returns a maximum (s, t)-flow of minimum cost.

networkx.algorithms.flow.network_simplex

network_simplex(*G*, *demand='demand'*, *capacity='capacity'*, *weight='weight'*)

Find a minimum cost flow satisfying all demands in digraph G.

This is a primal network simplex algorithm that uses the leaving arc rule to prevent cycling.

G is a digraph with edge costs and capacities and in which nodes have demand, i.e., they want to send or receive some amount of flow. A negative demand means that the node wants to send flow, a positive demand means that the node want to receive flow. A flow on the digraph G satisfies all demand if the net flow into each node is equal to the demand of that node.

Parameters

- **G** (*NetworkX graph*) – DiGraph on which a minimum cost flow satisfying all demands is to be found.
- **demand** (*string*) – Nodes of the graph G are expected to have an attribute demand that indicates how much flow a node wants to send (negative demand) or receive (positive demand). Note that the sum of the demands should be 0 otherwise the problem in not feasible. If this attribute is not present, a node is considered to have 0 demand. Default value: 'demand'.
- **capacity** (*string*) – Edges of the graph G are expected to have an attribute capacity that indicates how much flow the edge can support. If this attribute is not present, the edge is considered to have infinite capacity. Default value: 'capacity'.
- **weight** (*string*) – Edges of the graph G are expected to have an attribute weight that indicates the cost incurred by sending one unit of flow on that edge. If not present, the weight is considered to be 0. Default value: 'weight'.

Returns

- **flowCost** (*integer, float*) – Cost of a minimum cost flow satisfying all demands.
- **flowDict** (*dictionary*) – Dictionary of dictionaries keyed by nodes such that flowDict[u][v] is the flow edge (u, v).

Raises

- ***NetworkXError*** – This exception is raised if the input graph is not directed, not connected or is a multigraph.
- ***NetworkXUnfeasible*** – This exception is raised in the following situations:
  - The sum of the demands is not zero. Then, there is no flow satisfying all demands.
  - There is no flow satisfying all demand.
- ***NetworkXUnbounded*** – This exception is raised if the digraph G has a cycle of negative cost and infinite capacity. Then, the cost of a flow satisfying all demands is unbounded below.

Notes

This algorithm is not guaranteed to work if edge weights or demands are floating point numbers (overflows and roundoff errors can cause problems). As a workaround you can use integer numbers by multiplying the relevant edge attributes by a convenient constant factor (eg 100).

See also:

cost_of_flow(), *max_flow_min_cost()*, *min_cost_flow()*, *min_cost_flow_cost()*

Examples

A simple example of a min cost flow problem.

```
>>> import networkx as nx
>>> G = nx.DiGraph()
>>> G.add_node('a', demand=-5)
>>> G.add_node('d', demand=5)
>>> G.add_edge('a', 'b', weight=3, capacity=4)
>>> G.add_edge('a', 'c', weight=6, capacity=10)
>>> G.add_edge('b', 'd', weight=1, capacity=9)
>>> G.add_edge('c', 'd', weight=2, capacity=5)
>>> flowCost, flowDict = nx.network_simplex(G)
>>> flowCost
24
>>> flowDict
{'a': {'c': 1, 'b': 4}, 'c': {'d': 1}, 'b': {'d': 4}, 'd': {}}
```

The mincost flow algorithm can also be used to solve shortest path problems. To find the shortest path between two nodes u and v, give all edges an infinite capacity, give node u a demand of -1 and node v a demand a 1. Then run the network simplex. The value of a min cost flow will be the distance between u and v and edges carrying positive flow will indicate the path.

```
>>> G=nx.DiGraph()
>>> G.add_weighted_edges_from([('s', 'u' ,10), ('s' ,'x' ,5),
...                            ('u', 'v' ,1), ('u' ,'x' ,2),
...                            ('v', 'y' ,1), ('x' ,'u' ,3),
...                            ('x', 'v' ,5), ('x' ,'y' ,2),
...                            ('y', 's' ,7), ('y' ,'v' ,6)])
>>> G.add_node('s', demand = -1)
>>> G.add_node('v', demand = 1)
>>> flowCost, flowDict = nx.network_simplex(G)
>>> flowCost == nx.shortest_path_length(G, 's', 'v', weight='weight')
True
```

(continues on next page)

(continued from previous page)

```
>>> sorted([(u, v) for u in flowDict for v in flowDict[u] if flowDict[u][v] > 0])
[('s', 'x'), ('u', 'v'), ('x', 'u')]
>>> nx.shortest_path(G, 's', 'v', weight = 'weight')
['s', 'x', 'u', 'v']
```

It is possible to change the name of the attributes used for the algorithm.

```
>>> G = nx.DiGraph()
>>> G.add_node('p', spam=-4)
>>> G.add_node('q', spam=2)
>>> G.add_node('a', spam=-2)
>>> G.add_node('d', spam=-1)
>>> G.add_node('t', spam=2)
>>> G.add_node('w', spam=3)
>>> G.add_edge('p', 'q', cost=7, vacancies=5)
>>> G.add_edge('p', 'a', cost=1, vacancies=4)
>>> G.add_edge('q', 'd', cost=2, vacancies=3)
>>> G.add_edge('t', 'q', cost=1, vacancies=2)
>>> G.add_edge('a', 't', cost=2, vacancies=4)
>>> G.add_edge('d', 'w', cost=3, vacancies=4)
>>> G.add_edge('t', 'w', cost=4, vacancies=1)
>>> flowCost, flowDict = nx.network_simplex(G, demand='spam',
...                                         capacity='vacancies',
...                                         weight='cost')
>>> flowCost
37
>>> flowDict
{'a': {'t': 4}, 'd': {'w': 2}, 'q': {'d': 1}, 'p': {'q': 2, 'a': 2}, 't': {'q': 1,
↪ 'w': 1}, 'w': {}}
```

References

networkx.algorithms.flow.min_cost_flow_cost

min_cost_flow_cost(*G*, *demand='demand'*, *capacity='capacity'*, *weight='weight'*)

Find the cost of a minimum cost flow satisfying all demands in digraph G.

G is a digraph with edge costs and capacities and in which nodes have demand, i.e., they want to send or receive some amount of flow. A negative demand means that the node wants to send flow, a positive demand means that the node want to receive flow. A flow on the digraph G satisfies all demand if the net flow into each node is equal to the demand of that node.

Parameters

- **G** (*NetworkX graph*) – DiGraph on which a minimum cost flow satisfying all demands is to be found.
- **demand** (*string*) – Nodes of the graph G are expected to have an attribute demand that indicates how much flow a node wants to send (negative demand) or receive (positive demand). Note that the sum of the demands should be 0 otherwise the problem in not feasible. If this attribute is not present, a node is considered to have 0 demand. Default value: 'demand'.
- **capacity** (*string*) – Edges of the graph G are expected to have an attribute capacity that indicates how much flow the edge can support. If this attribute is not present, the edge is considered to have infinite capacity. Default value: 'capacity'.

- **weight** (*string*) – Edges of the graph G are expected to have an attribute weight that indicates the cost incurred by sending one unit of flow on that edge. If not present, the weight is considered to be 0. Default value: 'weight'.

Returns **flowCost** – Cost of a minimum cost flow satisfying all demands.

Return type integer, float

Raises

- ***NetworkXError*** – This exception is raised if the input graph is not directed or not connected.
- ***NetworkXUnfeasible*** – This exception is raised in the following situations:
  - The sum of the demands is not zero. Then, there is no flow satisfying all demands.
  - There is no flow satisfying all demand.
- ***NetworkXUnbounded*** – This exception is raised if the digraph G has a cycle of negative cost and infinite capacity. Then, the cost of a flow satisfying all demands is unbounded below.

See also:

cost_of_flow(), *max_flow_min_cost()*, *min_cost_flow()*, *network_simplex()*

Notes

This algorithm is not guaranteed to work if edge weights or demands are floating point numbers (overflows and roundoff errors can cause problems). As a workaround you can use integer numbers by multiplying the relevant edge attributes by a convenient constant factor (eg 100).

Examples

A simple example of a min cost flow problem.

```
>>> import networkx as nx
>>> G = nx.DiGraph()
>>> G.add_node('a', demand = -5)
>>> G.add_node('d', demand = 5)
>>> G.add_edge('a', 'b', weight = 3, capacity = 4)
>>> G.add_edge('a', 'c', weight = 6, capacity = 10)
>>> G.add_edge('b', 'd', weight = 1, capacity = 9)
>>> G.add_edge('c', 'd', weight = 2, capacity = 5)
>>> flowCost = nx.min_cost_flow_cost(G)
>>> flowCost
24
```

networkx.algorithms.flow.min_cost_flow

min_cost_flow(*G*, *demand='demand'*, *capacity='capacity'*, *weight='weight'*)

Returns a minimum cost flow satisfying all demands in digraph G.

G is a digraph with edge costs and capacities and in which nodes have demand, i.e., they want to send or receive some amount of flow. A negative demand means that the node wants to send flow, a positive demand means that the node want to receive flow. A flow on the digraph G satisfies all demand if the net flow into each node is equal to the demand of that node.

Parameters

- **G** (*NetworkX graph*) – DiGraph on which a minimum cost flow satisfying all demands is to be found.
- **demand** (*string*) – Nodes of the graph G are expected to have an attribute demand that indicates how much flow a node wants to send (negative demand) or receive (positive demand). Note that the sum of the demands should be 0 otherwise the problem in not feasible. If this attribute is not present, a node is considered to have 0 demand. Default value: 'demand'.
- **capacity** (*string*) – Edges of the graph G are expected to have an attribute capacity that indicates how much flow the edge can support. If this attribute is not present, the edge is considered to have infinite capacity. Default value: 'capacity'.
- **weight** (*string*) – Edges of the graph G are expected to have an attribute weight that indicates the cost incurred by sending one unit of flow on that edge. If not present, the weight is considered to be 0. Default value: 'weight'.

Returns **flowDict** – Dictionary of dictionaries keyed by nodes such that flowDict[u][v] is the flow edge (u, v).

Return type dictionary

Raises

- ***NetworkXError*** – This exception is raised if the input graph is not directed or not connected.
- ***NetworkXUnfeasible*** – This exception is raised in the following situations:
  - The sum of the demands is not zero. Then, there is no flow satisfying all demands.
  - There is no flow satisfying all demand.
- ***NetworkXUnbounded*** – This exception is raised if the digraph G has a cycle of negative cost and infinite capacity. Then, the cost of a flow satisfying all demands is unbounded below.

See also:

cost_of_flow(), *max_flow_min_cost()*, *min_cost_flow_cost()*, *network_simplex()*

Notes

This algorithm is not guaranteed to work if edge weights or demands are floating point numbers (overflows and roundoff errors can cause problems). As a workaround you can use integer numbers by multiplying the relevant edge attributes by a convenient constant factor (eg 100).

Examples

A simple example of a min cost flow problem.

```
>>> import networkx as nx
>>> G = nx.DiGraph()
>>> G.add_node('a', demand = -5)
>>> G.add_node('d', demand = 5)
>>> G.add_edge('a', 'b', weight = 3, capacity = 4)
>>> G.add_edge('a', 'c', weight = 6, capacity = 10)
>>> G.add_edge('b', 'd', weight = 1, capacity = 9)
```

(continues on next page)

(continued from previous page)

```
>>> G.add_edge('c', 'd', weight = 2, capacity = 5)
>>> flowDict = nx.min_cost_flow(G)
```

networkx.algorithms.flow.cost_of_flow

cost_of_flow(*G*, *flowDict*, *weight='weight'*)

Compute the cost of the flow given by flowDict on graph G.

Note that this function does not check for the validity of the flow flowDict. This function will fail if the graph G and the flow don't have the same edge set.

Parameters

- **G** (*NetworkX graph*) – DiGraph on which a minimum cost flow satisfying all demands is to be found.
- **weight** (*string*) – Edges of the graph G are expected to have an attribute weight that indicates the cost incurred by sending one unit of flow on that edge. If not present, the weight is considered to be 0. Default value: 'weight'.
- **flowDict** (*dictionary*) – Dictionary of dictionaries keyed by nodes such that flowDict[u][v] is the flow edge (u, v).

Returns **cost** – The total cost of the flow. This is given by the sum over all edges of the product of the edge's flow and the edge's weight.

Return type Integer, float

See also:

max_flow_min_cost(), *min_cost_flow()*, *min_cost_flow_cost()*, *network_simplex()*

Notes

This algorithm is not guaranteed to work if edge weights or demands are floating point numbers (overflows and roundoff errors can cause problems). As a workaround you can use integer numbers by multiplying the relevant edge attributes by a convenient constant factor (eg 100).

networkx.algorithms.flow.max_flow_min_cost

max_flow_min_cost(*G*, *s*, *t*, *capacity='capacity'*, *weight='weight'*)

Returns a maximum (s, t)-flow of minimum cost.

G is a digraph with edge costs and capacities. There is a source node s and a sink node t. This function finds a maximum flow from s to t whose total cost is minimized.

Parameters

- **G** (*NetworkX graph*) – DiGraph on which a minimum cost flow satisfying all demands is to be found.
- **s** (*node label*) – Source of the flow.
- **t** (*node label*) – Destination of the flow.
- **capacity** (*string*) – Edges of the graph G are expected to have an attribute capacity that indicates how much flow the edge can support. If this attribute is not present, the edge is considered to have infinite capacity. Default value: 'capacity'.

- **weight** (*string*) – Edges of the graph G are expected to have an attribute weight that indicates the cost incurred by sending one unit of flow on that edge. If not present, the weight is considered to be 0. Default value: 'weight'.

Returns **flowDict** – Dictionary of dictionaries keyed by nodes such that flowDict[u][v] is the flow edge (u, v).

Return type dictionary

Raises

- ***NetworkXError*** – This exception is raised if the input graph is not directed or not connected.
- ***NetworkXUnbounded*** – This exception is raised if there is an infinite capacity path from s to t in G. In this case there is no maximum flow. This exception is also raised if the digraph G has a cycle of negative cost and infinite capacity. Then, the cost of a flow is unbounded below.

See also:

cost_of_flow(), *min_cost_flow()*, *min_cost_flow_cost()*, *network_simplex()*

Notes

This algorithm is not guaranteed to work if edge weights or demands are floating point numbers (overflows and roundoff errors can cause problems). As a workaround you can use integer numbers by multiplying the relevant edge attributes by a convenient constant factor (eg 100).

Examples

```
>>> G = nx.DiGraph()
>>> G.add_edges_from([(1, 2, {'capacity': 12, 'weight': 4}),
...                   (1, 3, {'capacity': 20, 'weight': 6}),
...                   (2, 3, {'capacity': 6, 'weight': -3}),
...                   (2, 6, {'capacity': 14, 'weight': 1}),
...                   (3, 4, {'weight': 9}),
...                   (3, 5, {'capacity': 10, 'weight': 5}),
...                   (4, 2, {'capacity': 19, 'weight': 13}),
...                   (4, 5, {'capacity': 4, 'weight': 0}),
...                   (5, 7, {'capacity': 28, 'weight': 2}),
...                   (6, 5, {'capacity': 11, 'weight': 1}),
...                   (6, 7, {'weight': 8}),
...                   (7, 4, {'capacity': 6, 'weight': 6})])
>>> mincostFlow = nx.max_flow_min_cost(G, 1, 7)
>>> mincost = nx.cost_of_flow(G, mincostFlow)
>>> mincost
373
>>> from networkx.algorithms.flow import maximum_flow
>>> maxFlow = maximum_flow(G, 1, 7)[1]
>>> nx.cost_of_flow(G, maxFlow) >= mincost
True
>>> mincostFlowValue = (sum((mincostFlow[u][7] for u in G.predecessors(7)))
...                     - sum((mincostFlow[7][v] for v in G.successors(7))))
>>> mincostFlowValue == nx.maximum_flow_value(G, 1, 7)
True
```

CHAPTER

TEN

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.

10.1 Matplotlib

10.1.1 Matplotlib

Draw networks with matplotlib.

See also:

matplotlib http://matplotlib.org/

pygraphviz http://pygraphviz.github.io/

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 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, **kwargs)	Draw networkx graph with shell layout.

10.1.2 networkx.drawing.nx_pylab.draw

draw(*G*, *pos=None*, *ax=None*, ***kwds*)

Draw the graph G with Matplotlib.

Draw the graph as a simple representation with no node labels or edge labels and using the full Matplotlib figure area and no axis labels by default. See draw_networkx() for more full-featured drawing that allows title, axis labels etc.

Parameters

- **G** (*graph*) – A networkx graph
- **pos** (*dictionary, optional*) – A dictionary with nodes as keys and positions as values. If not specified a spring layout positioning will be computed. See *networkx.drawing.layout* for functions that compute node positions.
- **ax** (*Matplotlib Axes object, optional*) – Draw the graph in specified Matplotlib axes.
- **kwds** (*optional keywords*) – See networkx.draw_networkx() for a description of optional keywords.

Examples

```
>>> G = nx.dodecahedral_graph()
>>> nx.draw(G)
>>> nx.draw(G, pos=nx.spring_layout(G))  # use spring layout
```

See also:

draw_networkx(), *draw_networkx_nodes()*, *draw_networkx_edges()*, *draw_networkx_labels()*, *draw_networkx_edge_labels()*

Notes

This function has the same name as pylab.draw and pyplot.draw so beware when using `from networkx import *`

since you might overwrite the pylab.draw function.

With pyplot use

```
>>> import matplotlib.pyplot as plt
>>> import networkx as nx
>>> G = nx.dodecahedral_graph()
>>> nx.draw(G)  # networkx draw()
>>> plt.draw()  # pyplot draw()
```

Also see the NetworkX drawing examples at https://networkx.github.io/documentation/latest/auto_examples/index.html

10.1.3 networkx.drawing.nx_pylab.draw_networkx

draw_networkx(*G*, *pos=None*, *arrows=True*, *with_labels=True*, ***kwds*)

Draw the graph G using Matplotlib.

Draw the graph with Matplotlib with options for node positions, labeling, titles, and many other drawing features. See draw() for simple drawing without labels or axes.

Parameters

- **G** (*graph*) – A networkx graph
- **pos** (*dictionary, optional*) – A dictionary with nodes as keys and positions as values. If not specified a spring layout positioning will be computed. See *networkx.drawing.layout* for functions that compute node positions.
- **arrows** (*bool, optional (default=True)*) – For directed graphs, if True draw arrowheads. Note: Arrows will be the same color as edges.
- **arrowstyle** (*str, optional (default='-|>')*) – For directed graphs, choose the style of the arrowsheads. See :py:class: `matplotlib.patches.ArrowStyle` for more options.
- **arrowsize** (*int, optional (default=10)*) – For directed graphs, choose the size of the arrow head head's length and width. See :py:class: `matplotlib.patches.FancyArrowPatch` for attribute `mutation_scale` for more info.
- **with_labels** (*bool, optional (default=True)*) – Set to True to draw labels on the nodes.
- **ax** (*Matplotlib Axes object, optional*) – Draw the graph in the specified Matplotlib axes.
- **nodelist** (*list, optional (default G.nodes())*) – Draw only specified nodes
- **edgelist** (*list, optional (default=G.edges())*) – Draw only specified edges
- **node_size** (*scalar or array, optional (default=300)*) – Size of nodes. If an array is specified it must be the same length as nodelist.
- **node_color** (*color or array of colors (default='#1f78b4')*) – Node color. Can be a single color or a sequence of colors with the same length as nodelist. Color can be string, or rgb (or rgba) tuple of floats from 0-1. If numeric values are specified they will be mapped to colors using the cmap and vmin,vmax parameters. See matplotlib.scatter for more details.
- **node_shape** (*string, optional (default='o')*) – The shape of the node. Specification is as matplotlib.scatter marker, one of 'so^>v<dph8'.
- **alpha** (*float, optional (default=None)*) – The node and edge transparency
- **cmap** (*Matplotlib colormap, optional (default=None)*) – Colormap for mapping intensities of nodes
- **vmin,vmax** (*float, optional (default=None)*) – Minimum and maximum for node colormap scaling
- **linewidths** (*[None | scalar | sequence]*) – Line width of symbol border (default =1.0)
- **width** (*float, optional (default=1.0)*) – Line width of edges
- **edge_color** (*color or array of colors (default='k')*) – Edge color. Can be a single color or a sequence of colors with the same length as edgelist. Color can be string, or rgb (or rgba) tuple of floats from 0-1. If numeric values are specified they will be mapped to colors using the edge_cmap and edge_vmin,edge_vmax parameters.
- **edge_cmap** (*Matplotlib colormap, optional (default=None)*) – Colormap for mapping intensities of edges
- **edge_vmin,edge_vmax** (*floats, optional (default=None)*) – Minimum and maximum for edge colormap scaling
- **style** (*string, optional (default='solid')*) – Edge line style (solid|dashed|dotted,dashdot)

- **labels** (*dictionary, optional (default=None)*) – Node labels in a dictionary keyed by node of text labels
- **font_size** (*int, optional (default=12)*) – Font size for text labels
- **font_color** (*string, optional (default='k' black)*) – Font color string
- **font_weight** (*string, optional (default='normal')*) – Font weight
- **font_family** (*string, optional (default='sans-serif')*) – Font family
- **label** (*string, optional*) – Label for graph legend

Notes

For directed graphs, arrows are drawn at the head end. Arrows can be turned off with keyword arrows=False.

Examples

```
>>> G = nx.dodecahedral_graph()
>>> nx.draw(G)
>>> nx.draw(G, pos=nx.spring_layout(G))  # use spring layout
```

```
>>> import matplotlib.pyplot as plt
>>> limits = plt.axis('off')  # turn of axis
```

Also see the NetworkX drawing examples at https://networkx.github.io/documentation/latest/auto_examples/index.html

See also:

draw(), *draw_networkx_nodes()*, *draw_networkx_edges()*, *draw_networkx_labels()*, *draw_networkx_edge_labels()*

10.1.4 networkx.drawing.nx_pylab.draw_networkx_nodes

draw_networkx_nodes(*G*, *pos*, *nodelist=None*, *node_size=300*, *node_color='#1f78b4'*, *node_shape='o'*, *alpha=None*, *cmap=None*, *vmin=None*, *vmax=None*, *ax=None*, *linewidths=None*, *edgecolors=None*, *label=None*, ***kwds*)

Draw the nodes of the graph G.

This draws only the nodes of the graph G.

Parameters

- **G** (*graph*) – A networkx graph
- **pos** (*dictionary*) – A dictionary with nodes as keys and positions as values. Positions should be sequences of length 2.
- **ax** (*Matplotlib Axes object, optional*) – Draw the graph in the specified Matplotlib axes.
- **nodelist** (*list, optional*) – Draw only specified nodes (default G.nodes())
- **node_size** (*scalar or array*) – Size of nodes (default=300). If an array is specified it must be the same length as nodelist.

- **node_color** (*color or array of colors (default='#1f78b4')*) – Node color. Can be a single color or a sequence of colors with the same length as nodelist. Color can be string, or rgb (or rgba) tuple of floats from 0-1. If numeric values are specified they will be mapped to colors using the cmap and vmin,vmax parameters. See matplotlib.scatter for more details.
- **node_shape** (*string*) – The shape of the node. Specification is as matplotlib.scatter marker, one of 'so^>v<dph8' (default='o').
- **alpha** (*float or array of floats*) – The node transparency. This can be a single alpha value (default=None), in which case it will be applied to all the nodes of color. Otherwise, if it is an array, the elements of alpha will be applied to the colors in order (cycling through alpha multiple times if necessary).
- **cmap** (*Matplotlib colormap*) – Colormap for mapping intensities of nodes (default=None)
- **vmin,vmax** (*floats*) – Minimum and maximum for node colormap scaling (default=None)
- **linewidths** (*[None | scalar | sequence]*) – Line width of symbol border (default =1.0)
- **edgecolors** (*[None | scalar | sequence]*) – Colors of node borders (default = node_color)
- **label** (*[None| string]*) – Label for legend
- **min_source_margin** (*int, optional (default=0)*) – The minimum margin (gap) at the beginning of the edge at the source.
- **min_target_margin** (*int, optional (default=0)*) – The minimum margin (gap) at the end of the edge at the target.

Returns `PathCollection` of the nodes.

Return type matplotlib.collections.PathCollection

Examples

```
>>> G = nx.dodecahedral_graph()
>>> nodes = nx.draw_networkx_nodes(G, pos=nx.spring_layout(G))
```

Also see the NetworkX drawing examples at https://networkx.github.io/documentation/latest/auto_examples/index.html

See also:

draw(), *draw_networkx()*, *draw_networkx_edges()*, *draw_networkx_labels()*, *draw_networkx_edge_labels()*

10.1.5 networkx.drawing.nx_pylab.draw_networkx_edges

draw_networkx_edges(*G*, *pos*, *edgelist=None*, *width=1.0*, *edge_color='k'*, *style='solid'*, *alpha=None*, *arrowstyle='-|>'*, *arrowsize=10*, *edge_cmap=None*, *edge_vmin=None*, *edge_vmax=None*, *ax=None*, *arrows=True*, *label=None*, *node_size=300*, *nodelist=None*, *node_shape='o'*, *connectionstyle=None*, *min_source_margin=0*, *min_target_margin=0*, ***kwds*)

Draw the edges of the graph G.

This draws only the edges of the graph G.

Parameters

- **G** (*graph*) – A networkx graph

- **pos** (*dictionary*) – A dictionary with nodes as keys and positions as values. Positions should be sequences of length 2.
- **edgelist** (*collection of edge tuples*) – Draw only specified edges(default=G.edges())
- **width** (*float, or array of floats*) – Line width of edges (default=1.0)
- **edge_color** (*color or array of colors (default='k')*) – Edge color. Can be a single color or a sequence of colors with the same length as edgelist. Color can be string, or rgb (or rgba) tuple of floats from 0-1. If numeric values are specified they will be mapped to colors using the edge_cmap and edge_vmin,edge_vmax parameters.
- **style** (*string*) – Edge line style (default='solid') (solid|dashed|dotted,dashdot)
- **alpha** (*float*) – The edge transparency (default=None)
- **edge_ cmap** (*Matplotlib colormap*) – Colormap for mapping intensities of edges (default=None)
- **edge_vmin,edge_vmax** (*floats*) – Minimum and maximum for edge colormap scaling (default=None)
- **ax** (*Matplotlib Axes object, optional*) – Draw the graph in the specified Matplotlib axes.
- **arrows** (*bool, optional (default=True)*) – For directed graphs, if True draw arrowheads. Note: Arrows will be the same color as edges.
- **arrowstyle** (*str, optional (default='-|>')*) – For directed graphs, choose the style of the arrow heads. See :py:class: `matplotlib.patches.ArrowStyle` for more options.
- **arrowsize** (*int, optional (default=10)*) – For directed graphs, choose the size of the arrow head head's length and width. See :py:class: `matplotlib.patches.FancyArrowPatch` for attribute `mutation_scale` for more info.
- **connectionstyle** (*str, optional (default=None)*) – Pass the connectionstyle parameter to create curved arc of rounding radius rad. For example, connectionstyle='arc3,rad=0.2'. See :py:class: `matplotlib.patches.ConnectionStyle` and :py:class: `matplotlib.patches.FancyArrowPatch` for more info.
- **label** (*[None| string]*) – Label for legend
- **min_source_margin** (*int, optional (default=0)*) – The minimum margin (gap) at the beginning of the edge at the source.
- **min_target_margin** (*int, optional (default=0)*) – The minimum margin (gap) at the end of the edge at the target.

Returns

- *matplotlib.collection.LineCollection* – `LineCollection` of the edges
- *list of matplotlib.patches.FancyArrowPatch* – `FancyArrowPatch` instances of the directed edges
- *Depending whether the drawing includes arrows or not.*

Notes

For directed graphs, arrows are drawn at the head end. Arrows can be turned off with keyword arrows=False. Be sure to include `node_size` as a keyword argument; arrows are drawn considering the size of nodes.

Examples

```
>>> G = nx.dodecahedral_graph()
>>> edges = nx.draw_networkx_edges(G, pos=nx.spring_layout(G))
```

```
>>> G = nx.DiGraph()
>>> G.add_edges_from([(1, 2), (1, 3), (2, 3)])
>>> arcs = nx.draw_networkx_edges(G, pos=nx.spring_layout(G))
>>> alphas = [0.3, 0.4, 0.5]
>>> for i, arc in enumerate(arcs):  # change alpha values of arcs
...     arc.set_alpha(alphas[i])
```

Also see the NetworkX drawing examples at https://networkx.github.io/documentation/latest/auto_examples/index.html

See also:

draw(), *draw_networkx()*, *draw_networkx_nodes()*, *draw_networkx_labels()*, *draw_networkx_edge_labels()*

10.1.6 networkx.drawing.nx_pylab.draw_networkx_labels

draw_networkx_labels(*G*, *pos*, *labels=None*, *font_size=12*, *font_color='k'*, *font_family='sans-serif'*, *font_weight='normal'*, *alpha=None*, *bbox=None*, *ax=None*, ***kwds*)

Draw node labels on the graph G.

Parameters

- **G** (*graph*) – A networkx graph
- **pos** (*dictionary*) – A dictionary with nodes as keys and positions as values. Positions should be sequences of length 2.
- **labels** (*dictionary, optional (default=None)*) – Node labels in a dictionary keyed by node of text labels Node-keys in labels should appear as keys in `pos`. If needed use: `{n:lab for n,lab in labels.items() if n in pos}`
- **font_size** (*int*) – Font size for text labels (default=12)
- **font_color** (*string*) – Font color string (default='k' black)
- **font_family** (*string*) – Font family (default='sans-serif')
- **font_weight** (*string*) – Font weight (default='normal')
- **alpha** (*float or None*) – The text transparency (default=None)
- **ax** (*Matplotlib Axes object, optional*) – Draw the graph in the specified Matplotlib axes.

Returns `dict` of labels keyed on the nodes

Return type dict

Examples

```
>>> G = nx.dodecahedral_graph()
>>> labels = nx.draw_networkx_labels(G, pos=nx.spring_layout(G))
```

Also see the NetworkX drawing examples at https://networkx.github.io/documentation/latest/auto_examples/index.html

See also:

draw(), *draw_networkx()*, *draw_networkx_nodes()*, *draw_networkx_edges()*, *draw_networkx_edge_labels()*

10.1.7 networkx.drawing.nx_pylab.draw_networkx_edge_labels

draw_networkx_edge_labels(*G*, *pos*, *edge_labels=None*, *label_pos=0.5*, *font_size=10*, *font_color='k'*, *font_family='sans-serif'*, *font_weight='normal'*, *alpha=None*, *bbox=None*, *ax=None*, *rotate=True*, ***kwds*)

Draw edge labels.

Parameters

- **G** (*graph*) – A networkx graph
- **pos** (*dictionary*) – A dictionary with nodes as keys and positions as values. Positions should be sequences of length 2.
- **ax** (*Matplotlib Axes object, optional*) – Draw the graph in the specified Matplotlib axes.
- **alpha** (*float or None*) – The text transparency (default=None)
- **edge_labels** (*dictionary*) – Edge labels in a dictionary keyed by edge two-tuple of text labels (default=None). Only labels for the keys in the dictionary are drawn.
- **label_pos** (*float*) – Position of edge label along edge (0=head, 0.5=center, 1=tail)
- **font_size** (*int*) – Font size for text labels (default=12)
- **font_color** (*string*) – Font color string (default='k' black)
- **font_weight** (*string*) – Font weight (default='normal')
- **font_family** (*string*) – Font family (default='sans-serif')
- **bbox** (*Matplotlib bbox*) – Specify text box shape and colors.
- **clip_on** (*bool*) – Turn on clipping at axis boundaries (default=True)

Returns dict of labels keyed on the edges

Return type dict

Examples

```
>>> G = nx.dodecahedral_graph()
>>> edge_labels = nx.draw_networkx_edge_labels(G, pos=nx.spring_layout(G))
```

Also see the NetworkX drawing examples at https://networkx.github.io/documentation/latest/auto_examples/index.html

See also:

draw(), *draw_networkx()*, *draw_networkx_nodes()*, *draw_networkx_edges()*, *draw_networkx_labels()*

10.1.8 networkx.drawing.nx_pylab.draw_circular

draw_circular(*G*, ***kwargs*)
Draw the graph G with a circular layout.

Parameters

- **G** (*graph*) – A networkx graph
- **kwargs** (*optional keywords*) – See networkx.draw_networkx() for a description of optional keywords, with the exception of the pos parameter which is not used by this function.

10.1.9 networkx.drawing.nx_pylab.draw_kamada_kawai

draw_kamada_kawai(*G*, ***kwargs*)
Draw the graph G with a Kamada-Kawai force-directed layout.

Parameters

- **G** (*graph*) – A networkx graph
- **kwargs** (*optional keywords*) – See networkx.draw_networkx() for a description of optional keywords, with the exception of the pos parameter which is not used by this function.

10.1.10 networkx.drawing.nx_pylab.draw_planar

draw_planar(*G*, ***kwargs*)
Draw a planar networkx graph with planar layout.

Parameters

- **G** (*graph*) – A planar networkx graph
- **kwargs** (*optional keywords*) – See networkx.draw_networkx() for a description of optional keywords, with the exception of the pos parameter which is not used by this function.

10.1.11 networkx.drawing.nx_pylab.draw_random

draw_random(*G*, ***kwargs*)
Draw the graph G with a random layout.

Parameters

- **G** (*graph*) – A networkx graph
- **kwargs** (*optional keywords*) – See networkx.draw_networkx() for a description of optional keywords, with the exception of the pos parameter which is not used by this function.

10.1.12 networkx.drawing.nx_pylab.draw_spectral

draw_spectral(*G*, ***kwargs*)
Draw the graph G with a spectral 2D layout.

Using the unnormalized Laplacion, the layout shows possible clusters of nodes which are an approximation of the ratio cut. The positions are the entries of the second and third eigenvectors corresponding to the ascending eigenvalues starting from the second one.

Parameters

- **G** (*graph*) – A networkx graph
- **kwargs** (*optional keywords*) – See networkx.draw_networkx() for a description of optional keywords, with the exception of the pos parameter which is not used by this function.

10.1.13 networkx.drawing.nx_pylab.draw_spring

draw_spring(*G*, ***kwargs*)
Draw the graph G with a spring layout.

Parameters

- **G** (*graph*) – A networkx graph
- **kwargs** (*optional keywords*) – See networkx.draw_networkx() for a description of optional keywords, with the exception of the pos parameter which is not used by this function.

10.1.14 networkx.drawing.nx_pylab.draw_shell

draw_shell(*G*, ***kwargs*)
Draw networkx graph with shell layout.

Parameters

- **G** (*graph*) – A networkx graph
- **kwargs** (*optional keywords*) – See networkx.draw_networkx() for a description of optional keywords, with the exception of the pos parameter which is not used by this function.

10.2 Graphviz AGraph (dot)

10.2.1 Graphviz AGraph

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/

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.

Continued on next page