

---

# retworkx Documentation

*Release 0.8.0*

**Matthew Treinish**

**May 24, 2021**



# CONTENTS

<b>1</b>	<b>networkx</b>	<b>3</b>
1.1	Installing networkx . . . . .	3
1.1.1	Installing on a platform without precompiled binaries . . . . .	3
1.2	Building from source . . . . .	4
1.2.1	Develop Mode . . . . .	4
1.3	Using networkx . . . . .	4
<b>2</b>	<b>Networkx API Reference</b>	<b>7</b>
2.1	Graph Classes . . . . .	7
2.1.1	networkx.PyGraph . . . . .	7
2.1.2	networkx.PyDiGraph . . . . .	20
2.1.3	networkx.PyDAG . . . . .	37
2.2	Generators . . . . .	54
2.2.1	networkx.generators.cycle_graph . . . . .	55
2.2.2	networkx.generators.directed_cycle_graph . . . . .	56
2.2.3	networkx.generators.path_graph . . . . .	58
2.2.4	networkx.generators.directed_path_graph . . . . .	60
2.2.5	networkx.generators.star_graph . . . . .	62
2.2.6	networkx.generators.directed_star_graph . . . . .	63
2.2.7	networkx.generators.mesh_graph . . . . .	65
2.2.8	networkx.generators.directed_mesh_graph . . . . .	66
2.2.9	networkx.generators.grid_graph . . . . .	68
2.2.10	networkx.generators.directed_grid_graph . . . . .	69
2.3	Random Circuit Functions . . . . .	71
2.3.1	networkx.directed_gnp_random_graph . . . . .	71
2.3.2	networkx.undirected_gnp_random_graph . . . . .	72
2.3.3	networkx.directed_gnm_random_graph . . . . .	72
2.3.4	networkx.undirected_gnm_random_graph . . . . .	73
2.4	Algorithm Functions . . . . .	73
2.4.1	Specific Graph Type Methods . . . . .	73
2.4.2	Universal Functions . . . . .	96
2.5	Exceptions . . . . .	102
2.5.1	networkx.InvalidNode . . . . .	103
2.5.2	networkx.DAGWouldCycle . . . . .	103
2.5.3	networkx.NoEdgeBetweenNodes . . . . .	103
2.5.4	networkx.DAGHasCycle . . . . .	104
2.5.5	networkx.NoSuitableNeighbors . . . . .	104
2.5.6	networkx.NoPathFound . . . . .	104
2.5.7	networkx.NullGraph . . . . .	105
2.6	Return Iterator Types . . . . .	105

2.6.1	retworkx.BFSSuccessors	105
2.6.2	retworkx.NodeIndices	106
2.6.3	retworkx.EdgeList	107
2.6.4	retworkx.WeightedEdgeList	107
<b>3</b>	<b>Release Notes</b>	<b>109</b>
3.1	0.8.0	109
3.1.1	Prelude	109
3.1.2	New Features	109
3.1.3	Bug Fixes	112
3.2	0.7.2	113
3.2.1	Bug Fixes	113
<b>4</b>	<b>0.7.1</b>	<b>115</b>
<b>5</b>	<b>0.7.0</b>	<b>117</b>
5.1	New Features	117
5.2	Upgrade Notes	117
5.3	Fixes	118
<b>6</b>	<b>0.6.0</b>	<b>119</b>
6.1	New Features	119
6.2	Upgrade Notes	120
6.3	Fixes	120
<b>7</b>	<b>0.5.0</b>	<b>121</b>
7.1	New Features	121
7.2	Fixes	122
<b>8</b>	<b>0.4.0</b>	<b>123</b>
8.1	New Features	123
8.2	Upgrade Notes	124
8.3	Fixes	124
<b>9</b>	<b>Contributing</b>	<b>125</b>
9.1	Contributing to retworkx	125
9.1.1	Tests	125
9.1.2	Style	125
9.1.3	Building documentation	126
9.1.4	Release Notes	126
<b>10</b>	<b>retworkx for networkx users</b>	<b>129</b>
10.1	Some Key Differences	129
10.2	Graph Data and Attributes	130
10.2.1	Nodes	130
10.2.2	Edges	131
10.2.3	Attributes	131
10.2.4	Examining elements of a graph	131
10.3	API Equivalents	132
10.3.1	Class Constructors	132
10.3.2	Graph Modifiers	132
10.4	Functionality Gaps	133
<b>Index</b>		<b>135</b>

**Warning:** The documentation for retworkx has migrated to:

<https://qiskit.org/documentation/retworkx>

These docs will no longer be updated.

Contents:

**Warning:** The documentation for retworkx has migrated to:

<https://qiskit.org/documentation/retworkx>

These docs will no longer be updated.



## NETWORKX

- You can see the full rendered docs at: <https://networkx.readthedocs.io/en/latest/index.html>

networkx is a general purpose graph library for python3 written in Rust to take advantage of the performance and safety that Rust provides. It was built as a replacement for qiskit's previous (and current) networkx usage (hence the name) but is designed to provide a high performance general purpose graph library for any python application. The project was originally started to build a faster directed graph to use as the underlying data structure for the DAG at the center of qiskit-terra's transpiler, but it has since grown to cover all the graph usage in Qiskit and other applications.

### 1.1 Installing networkx

networkx is published on pypi so on x86\_64, i686, ppc64le, s390x, and aarch64 Linux systems, x86\_64 on Mac OSX, and 32 and 64 bit Windows installing is as simple as running:

```
pip install networkx
```

This will install a precompiled version of networkx into your python environment.

#### 1.1.1 Installing on a platform without precompiled binaries

If there are no precompiled binaries published for your system you'll have to build the package from source. However, to be able to build the package from the published source package you need to have rust  $\geq 1.39$  installed (and also cargo which is normally included with rust) You can use rustup (a cross platform installer for rust) to make this simpler, or rely on other installation methods. A source package is also published on pypi, so you still can also run the above pip command to install it. Once you have rust properly installed, running:

```
pip install networkx
```

will build networkx for your local system from the source package and install it just as it would if there was a prebuilt binary available.

## 1.2 Building from source

The first step for building retworkx from source is to clone it locally with:

```
git clone https://github.com/Qiskit/retworkx.git
```

retworkx uses `PyO3` and `setuptools-rust` to build the python interface, which enables using standard python tooling to work. So, assuming you have rust installed, you can easily install retworkx into your python environment using `pip`. Once you have a local clone of the repo, change your current working directory to the root of the repo. Then, you can install retworkx into your python env with:

```
pip install .
```

Assuming your current working directory is still the root of the repo. Otherwise you can run:

```
pip install $PATH_TO_REPO_ROOT
```

which will install it the same way. Then retworkx is installed in your local python environment. There are 2 things to note when doing this though, first if you try to run python from the repo root using this method it will not work as you expect. There is a name conflict in the repo root because of the local python package shim used in building the package. Simply run your python scripts or programs using retworkx outside of the repo root. The second issue is that any local changes you make to the rust code will not be reflected live in your python environment, you'll need to recompile retworkx by rerunning `pip install` to have any changes reflected in your python environment.

### 1.2.1 Develop Mode

If you'd like to build retworkx in debug mode and use an interactive debugger while working on a change you can use `python setup.py develop` to build and install retworkx in develop mode. This will build retworkx without optimizations and include debuginfo which can be handy for debugging. Do note that installing retworkx this way will be significantly slower than using `pip install` and should only be used for debugging/development.

It's worth noting that `pip install -e` does not work, as it will link the python packaging shim to your python environment but not build the retworkx binary. If you want to build retworkx in debug mode you have to use `python setup.py develop`.

## 1.3 Using retworkx

Once you have retworkx installed you can use it by importing retworkx. All the functions and graph classes are off the root of the package. For example, building a DAG and adding 2 nodes with an edge between them would be:

```
import retworkx

my_dag = retworkx.PyDAG(cycle_check=True)
# add_node(), add_child(), and add_parent() return the node index
# The sole argument here can be any python object
root_node = my_dag.add_node("MyRoot")
# The second and third arguments can be any python object
my_dag.add_child(root_node, "AChild", ["EdgeData"])
```



**Warning:** The documentation for retworkx has migrated to:

<https://qiskit.org/documentation/retworkx>

These docs will no longer be updated.



## NETWORKX API REFERENCE

### 2.1 Graph Classes

<code>networkx.Graph([multigraph])</code>	A class for creating undirected graphs
<code>networkx.DiGraph([check_cycle, multigraph])</code>	A class for creating directed graphs
<code>networkx.DAG([check_cycle, multigraph])</code>	A class for creating direct acyclic graphs.

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

#### 2.1.1 networkx.Graph

**class Graph**(*multigraph=True, /*)

A class for creating undirected graphs

The Graph class is used to create an undirected graph. It can be a multigraph (have multiple edges between nodes). Each node and edge (although rarely used for edges) is indexed by an integer id. Additionally, each node and edge contains an arbitrary Python object as a weight/data payload. You can use the index for access to the data payload as in the following example:

```
import networkx

graph = networkx.Graph()
data_payload = "An arbitrary Python object"
node_index = graph.add_node(data_payload)
print("Node Index: %s" % node_index)
print(graph[node_index])
```

```
Node Index: 0
An arbitrary Python object
```

The DiGraph implements the Python mapping protocol for nodes so in addition to access you can also update the data payload with:

```
import networkx

graph = networkx.PyGraph()
data_payload = "An arbitrary Python object"
node_index = graph.add_node(data_payload)
graph[node_index] = "New Payload"
print("Node Index: %s" % node_index)
print(graph[node_index])
```

```
Node Index: 0
New Payload
```

**Parameters** `multigraph` (*bool*) – When this is set to `False` the created `PyGraph` object will not be a multigraph (which is the default behavior). When `False` if parallel edges are added the weight/weight from that method call will be used to update the existing edge in place.

`__init__()`

Initialize self. See `help(type(self))` for accurate signature.

## Methods

<code>__init__()</code>	Initialize self.
<code>add_edge(node_a, node_b, edge, /)</code>	Add an edge between 2 nodes.
<code>add_edges_from(obj_list, /)</code>	Add new edges to the graph.
<code>add_edges_from_no_data(obj_list, /)</code>	Add new edges to the graph without python data.
<code>add_node(obj, /)</code>	Add a new node to the graph.
<code>add_nodes_from(obj_list, /)</code>	Add new nodes to the graph.
<code>adj(node, /)</code>	Get the index and data for the neighbors of a node.
<code>compose(other, node_map, /[, node_map_func, ...])</code>	Add another <code>PyGraph</code> object into this <code>PyGraph</code>
<code>degree(node, /)</code>	Get the degree for a node
<code>edge_list()</code>	Get edge list
<code>edges()</code>	Return a list of all edge data.
<code>extend_from_edge_list(edge_list, /)</code>	Extend graph from an edge list
<code>extend_from_weighted_edge_list(edge_list, /)</code>	Extend graph from a weighted edge list
<code>from_adjacency_matrix(matrix, /)</code>	Create a new <code>PyGraph</code> object from an adjacency matrix
<code>get_all_edge_data(node_a, node_b, /)</code>	Return the edge data for all the edges between 2 nodes.
<code>get_edge_data(node_a, node_b, /)</code>	Return the edge data for the edge between 2 nodes.
<code>get_node_data(node, /)</code>	Return the node data for a given node index
<code>has_edge(node_a, node_b, /)</code>	Return <code>True</code> if there is an edge between <code>node_a</code> to <code>node_b</code> .
<code>neighbors(node, /)</code>	Get the neighbors of a node.
<code>node_indexes()</code>	Return a list of all node indexes.
<code>nodes()</code>	Return a list of all node data.
<code>read_edge_list(path, /[, comment, delimitator])</code>	Read an edge list file and create a new <code>PyGraph</code> object from the contents
<code>remove_edge(node_a, node_b, /)</code>	Remove an edge between 2 nodes.
<code>remove_edge_from_index(edge, /)</code>	Remove an edge identified by the provided index

continues on next page

Table 2 – continued from previous page

<code>remove_edges_from(index_list, /)</code>	Remove edges from the graph.
<code>remove_node(node, /)</code>	Remove a node from the graph.
<code>remove_nodes_from(index_list, /)</code>	Remove nodes from the graph.
<code>subgraph(nodes, /)</code>	Return a new PyGraph object for a subgraph of this graph
<code>to_dot([node_attr, edge_attr, graph_attr, ...])</code>	Generate a dot file from the graph
<code>update_edge(source, target, /, edge)</code>	Update an edge's weight/payload in place
<code>update_edge_by_index(source, target, /, edge)</code>	Update an edge's weight/data payload in place by the edge index
<code>weighted_edge_list()</code>	Get edge list with weights

## Attributes

<code>multigraph</code>	Whether the graph is a multigraph (allows multiple edges between nodes) or not
-------------------------	--

### `add_edge(node_a, node_b, edge, /)`

Add an edge between 2 nodes.

If `multigraph` is `False` and an edge already exists between `node_a` and `node_b` the weight/payload of that existing edge will be updated to be `edge`.

#### Parameters

- `node_a (int)` – Index of the parent node
- `node_b (int)` – Index of the child node
- `edge` – The object to set as the data for the edge. It can be any python object.

**Returns** The edge index for the newly created (or updated in the case of an existing edge with `multigraph=False`) edge.

**Return type** `int`

### `add_edges_from(obj_list, /)`

Add new edges to the graph.

**Parameters** `obj_list (list)` – A list of tuples of the form `(node_a, node_b, obj)` to attach to the graph. `node_a` and `node_b` are integer indexes describing where an edge should be added, and `obj` is the python object for the edge data.

If `multigraph` is `False` and an edge already exists between `node_a` and `node_b` the weight/payload of that existing edge will be updated to be `edge`. This will occur in order from `obj_list` so if there are multiple parallel edges in `obj_list` the last entry will be used.

**Returns** A list of `int` indices of the newly created edges

**Return type** `list`

### `add_edges_from_no_data(obj_list, /)`

Add new edges to the graph without python data.

**Parameters** `obj_list (list)` – A list of tuples of the form `(parent, child)` to attach to the graph. `parent` and `child` are integer indexes describing where an edge should be added. Unlike `add_edges_from()` there is no data payload and when the edge is created `None` will be used.

If *multigraph* is *False* and an edge already exists between *node\_a* and *node\_b* the weight/payload of that existing edge will be updated to be *None*.

**Returns** A list of int indices of the newly created edges

**Return type** list

**add\_node**(*obj*, /)

Add a new node to the graph.

**Parameters** *obj* – The python object to attach to the node

**Returns** The index of the newly created node

**Return type** int

**add\_nodes\_from**(*obj\_list*, /)

Add new nodes to the graph.

**Parameters** *obj\_list* (*list*) – A list of python object to attach to the graph.

**Returns** *indices* A list of int indices of the newly created nodes

**Return type** *NodeIndices*

**adj**(*node*, /)

Get the index and data for the neighbors of a node.

This will return a dictionary where the keys are the node indexes of the adjacent nodes (inbound or out-bound) and the value is the edge data objects between that adjacent node and the provided node. Note, that in the case of multigraphs only a single edge data object will be returned

**Parameters** *node* (*int*) – The index of the node to get the neighbors

**Returns** *neighbors* A dictionary where the keys are node indexes and the value is the edge data object for all nodes that share an edge with the specified node.

**Return type** dict

**compose**(*other*, *node\_map*, /, *node\_map\_func*=None, *edge\_map\_func*=None)

Add another PyGraph object into this PyGraph

**Parameters**

- **other** (*PyGraph*) – The other PyGraph object to add onto this graph.
- **node\_map** (*dict*) – A dictionary mapping node indexes from this PyGraph object to node indexes in the other PyGraph object. The keys are a node index in this graph and the value is a tuple of the node index in the other graph to add an edge to and the weight of that edge. For example:

```
{
  1: (2, "weight"),
  2: (4, "weight2")
}
```

- **node\_map\_func** – An optional python callable that will take in a single node weight/data object and return a new node weight/data object that will be used when adding a node from other onto this graph.
- **edge\_map\_func** – An optional python callable that will take in a single edge weight/data object and return a new edge weight/data object that will be used when adding an edge from other onto this graph.

**Returns** `new_node_ids`: A dictionary mapping node index from the other PyGraph to the equivalent node index in this PyDAG after they've been combined

**Return type** `dict`

For example, start by building a graph:

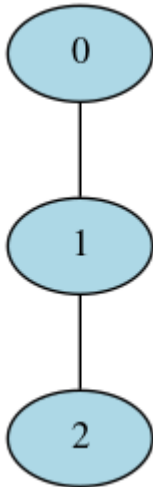
```
import os
import tempfile

import pydot
from PIL import Image

import networkx

# Build first graph and visualize:
graph = networkx.PyGraph()
node_a, node_b, node_c = graph.add_nodes_from(['A', 'B', 'C'])
graph.add_edges_from_no_data([(node_a, node_b), (node_b, node_c)])
dot_str = graph.to_dot(
    lambda node: dict(
        color='black', fillcolor='lightblue', style='filled'))
dot = pydot.graph_from_dot_data(dot_str)[0]

with tempfile.TemporaryDirectory() as tmpdirname:
    tmp_path = os.path.join(tmpdirname, 'graph.png')
    dot.write_png(tmp_path)
    image = Image.open(tmp_path)
    os.remove(tmp_path)
image
```



Then build a second one:

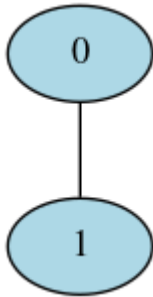
```
# Build second graph and visualize:
other_graph = networkx.PyGraph()
node_d, node_e = other_graph.add_nodes_from(['D', 'E'])
other_graph.add_edge(node_d, node_e, None)
dot_str = other_graph.to_dot(
```

(continues on next page)

(continued from previous page)

```
lambda node: dict(
    color='black', fillcolor='lightblue', style='filled'))
dot = pydot.graph_from_dot_data(dot_str)[0]

with tempfile.TemporaryDirectory() as tmpdirname:
    tmp_path = os.path.join(tmpdirname, 'other_graph.png')
    dot.write_png(tmp_path)
    image = Image.open(tmp_path)
    os.remove(tmp_path)
image
```

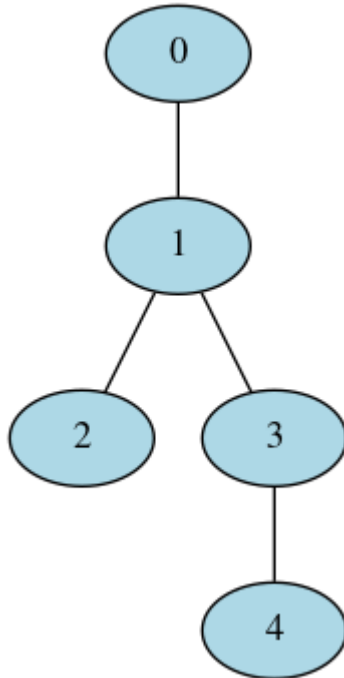


Finally compose the other\_graph onto graph

```
node_map = {node_b: (node_d, 'B to D')}
graph.compose(other_graph, node_map)
dot_str = graph.to_dot(
    lambda node: dict(
        color='black', fillcolor='lightblue', style='filled'))
dot = pydot.graph_from_dot_data(dot_str)[0]

with tempfile.TemporaryDirectory() as tmpdirname:
    tmp_path = os.path.join(tmpdirname, 'combined_graph.png')
    dot.write_png(tmp_path)
    image = Image.open(tmp_path)
    os.remove(tmp_path)
image
```





**degree**(*node*, /)

Get the degree for a node

**Parameters** *node* (*int*) – The index of the node to find the inbound degree of

**Returns** *degree* The inbound degree for the specified node

**Return type** *int*

**edge\_list**()

Get edge list

Returns a list of tuples of the form (*source*, *target*) where *source* and *target* are the node indices.

**Returns** An edge list with weights

**Return type** *EdgeList*

**edges**()

Return a list of all edge data.

**Returns** A list of all the edge data objects in the graph

**Return type** *list*

**extend\_from\_edge\_list**(*edge\_list*, /)

Extend graph from an edge list

This method differs from [add\\_edges\\_from\\_no\\_data\(\)](#) in that it will add nodes if a node index is not present in the edge list.

If *multigraph* is *False* and an edge already exists between *node\_a* and *node\_b* the weight/payload of that existing edge will be updated to be *None*.

**Parameters** *edge\_list* (*list*) – A list of tuples of the form (*source*, *target*) where *source* and *target* are integer node indices. If the node index is not present in the graph, nodes will be added (with a node weight of *None*) to that index.

**extend\_from\_weighted\_edge\_list**(*edge\_list*, /)

Extend graph from a weighted edge list

This method differs from *add\_edges\_from()* in that it will add nodes if a node index is not present in the edge list.

If *multigraph* is *False* and an edge already exists between *node\_a* and *node\_b* the weight/payload of that existing edge will be updated to be *edge*. This will occur in order from *obj\_list* so if there are multiple parallel edges in *obj\_list* the last entry will be used.

**Parameters** *edge\_list* (*list*) – A list of tuples of the form (*source*, *target*, *weight*) where *source* and *target* are integer node indices. If the node index is not present in the graph, nodes will be added (with a node weight of *None*) to that index.

**static from\_adjacency\_matrix**(*matrix*, /)

Create a new *PyGraph* object from an adjacency matrix

This method can be used to construct a new *PyGraph* object from an input adjacency matrix. The node weights will be the index from the matrix. The edge weights will be a float value of the value from the matrix.

**Parameters** *matrix* (*ndarray*) – The input numpy array adjacency matrix to create a new *PyGraph* object from. It must be a 2 dimensional array and be a *float/np.float64* data type.

**Returns** A new graph object generated from the adjacency matrix

**Return type** *PyGraph*

**get\_all\_edge\_data**(*node\_a*, *node\_b*, /)

Return the edge data for all the edges between 2 nodes.

**Parameters**

- **node\_a** (*int*) – The index for the first node
- **node\_b** (*int*) – The index for the second node

**Returns** A list with all the data objects for the edges between nodes

**Return type** *list*

**Raises** *NoEdgeBetweenNodes* – When there is no edge between nodes

**get\_edge\_data**(*node\_a*, *node\_b*, /)

Return the edge data for the edge between 2 nodes.

Note if there are multiple edges between the nodes only one will be returned. To get all edge data objects use *get\_all\_edge\_data()*

**Parameters**

- **node\_a** (*int*) – The index for the first node
- **node\_b** (*int*) – The index for the second node

**Returns** The data object set for the edge

**Raises** *NoEdgeBetweenNodes* – when there is no edge between the provided nodes

**get\_node\_data**(*node*, /)

Return the node data for a given node index

**Parameters** *node* (*int*) – The index for the node

**Returns** The data object set for that node

**Raises** **IndexError** – when an invalid node index is provided

**has\_edge**(*node\_a*, *node\_b*, /)

Return True if there is an edge between *node\_a* to *node\_b*.

**Parameters**

- **node\_a** (*int*) – The node index to check for an edge between
- **node\_b** (*int*) – The node index to check for an edge between

**Returns** True if there is an edge false if there is no edge

**Return type** bool

**multigraph**

Whether the graph is a multigraph (allows multiple edges between nodes) or not

If set to `False` multiple edges between nodes are not allowed and calls that would add a parallel edge will instead update the existing edge

**neighbors**(*node*, /)

Get the neighbors of a node.

This will return a list of neighbor node indices

**Parameters** **node** (*int*) – The index of the node to get the neighbors of

**Returns** A list of the neighbor node indices

**Return type** *NodeIndices*

**node\_indexes**()

Return a list of all node indexes.

**Returns** A list of all the node indexes in the graph

**Return type** *NodeIndices*

**nodes**()

Return a list of all node data.

**Returns** A list of all the node data objects in the graph

**Return type** list

**static read\_edge\_list**(*path*, /, *comment=None*, *delimiter=None*)

Read an edge list file and create a new `PyGraph` object from the contents

The expected format for the edge list file is a line separated list of delimited node ids. If there are more than 3 elements on a line the 3rd on will be treated as a string weight for the edge

**Parameters**

- **path** (*str*) – The path of the file to open
- **comment** (*str*) – Optional character to use as a comment by default there are no comment characters
- **delimiter** (*str*) – Optional character to use as a delimiter by default any whitespace will be used

For example:

```
import os
import tempfile

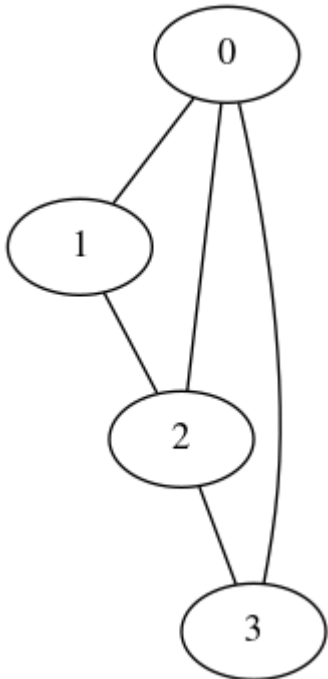
from PIL import Image
import pydot

import networkx

with tempfile.NamedTemporaryFile('wt') as fd:
    path = fd.name
    fd.write('0 1\n')
    fd.write('0 2\n')
    fd.write('0 3\n')
    fd.write('1 2\n')
    fd.write('2 3\n')
    fd.flush()
    graph = networkx.PyGraph.read_edge_list(path)

# Draw graph
dot = pydot.graph_from_dot_data(graph.to_dot())[0]

with tempfile.TemporaryDirectory() as tmpdirname:
    tmp_path = os.path.join(tmpdirname, 'dag.png')
    dot.write_png(tmp_path)
    image = Image.open(tmp_path)
    os.remove(tmp_path)
image
```



`remove_edge(node_a, node_b, /)`  
Remove an edge between 2 nodes.

Note if there are multiple edges between the specified nodes only one will be removed.

**Parameters**

- **parent** (*int*) – The index for the parent node.
- **child** (*int*) – The index of the child node.

**Raises** *NoEdgeBetweenNodes* – If there are no edges between the nodes specified

**remove\_edge\_from\_index**(*edge*, /)

Remove an edge identified by the provided index

**Parameters** **edge** (*int*) – The index of the edge to remove

**remove\_edges\_from**(*index\_list*, /)

Remove edges from the graph.

Note if there are multiple edges between the specified nodes only one will be removed.

**Parameters** **index\_list** (*list*) – A list of node index pairs to remove from the graph

**remove\_node**(*node*, /)

Remove a node from the graph.

**Parameters** **node** (*int*) – The index of the node to remove. If the index is not present in the graph it will be ignored and this function will have no effect.

**remove\_nodes\_from**(*index\_list*, /)

Remove nodes from the graph.

If a node index in the list is not present in the graph it will be ignored.

**Parameters** **index\_list** (*list*) – A list of node indices to remove from the the graph

**subgraph**(*nodes*, /)

Return a new PyGraph object for a subgraph of this graph

**Parameters** **nodes** (*list*) – A list of node indices to generate the subgraph from. If a node index is included that is not present in the graph it will silently be ignored.

**Returns** A new PyGraph object representing a subgraph of this graph. It is worth noting that node and edge weight/data payloads are passed by reference so if you update (not replace) an object used as the weight in graph or the subgraph it will also be updated in the other.

**Return type** *PyGraph*

**to\_dot**(*node\_attr=None*, *edge\_attr=None*, *graph\_attr=None*, *filename=None*)

Generate a dot file from the graph

**Parameters**

- **node\_attr** – A callable that will take in a node data object and return a dictionary of attributes to be associated with the node in the dot file. The key and value of this dictionary **must** be a string. If they're not strings networkx will raise TypeError (unfortunately without an error message because of current limitations in the PyO3 type checking)
- **edge\_attr** – A callable that will take in an edge data object and return a dictionary of attributes to be associated with the node in the dot file. The key and value of this dictionary **must** be a string. If they're not strings networkx will raise TypeError (unfortunately without an error message because of current limitations in the PyO3 type checking)
- **graph\_attr** (*dict*) – An optional dictionary that specifies any graph attributes for the output dot file. The key and value of this dictionary **must** be a string. If they're not strings

networkx will raise `TypeError` (unfortunately without an error message because of current limitations in the PyO3 type checking)

- **filename** (*str*) – An optional path to write the dot file to if specified there is no return from the function

**Returns** A string with the dot file contents if filename is not specified.

**Return type** *str*

Using this method enables you to leverage graphviz to visualize a `networkx.PyGraph` object. For example:

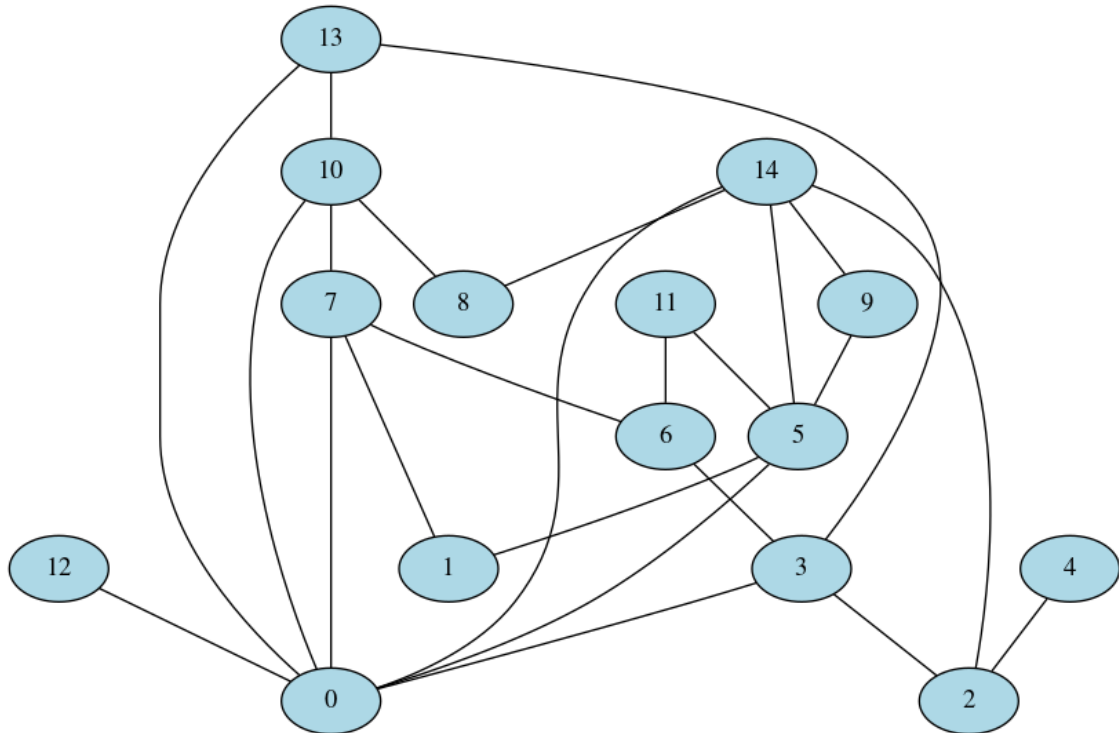
```
import os
import tempfile

import pydot
from PIL import Image

import networkx

graph = networkx.undirected_gnp_random_graph(15, .25)
dot_str = graph.to_dot(
    lambda node: dict(
        color='black', fillcolor='lightblue', style='filled'))
dot = pydot.graph_from_dot_data(dot_str)[0]

with tempfile.TemporaryDirectory() as tmpdirname:
    tmp_path = os.path.join(tmpdirname, 'dag.png')
    dot.write_png(tmp_path)
    image = Image.open(tmp_path)
    os.remove(tmp_path)
image
```



**update\_edge**(*source*, *target*, /, *edge*)

Update an edge's weight/payload in place

If there are parallel edges in the graph only one edge will be updated. if you need to update a specific edge or need to ensure all parallel edges get updated you should use `update_edge_by_index()` instead.

**Parameters**

- **source** (*int*) – The index for the first node
- **target** (*int*) – The index for the second node

**Raises** `NoEdgeBetweenNodes` – When there is no edge between nodes

**update\_edge\_by\_index**(*source*, *target*, /, *edge*)

Update an edge's weight/data payload in place by the edge index

**Parameters**

- **edge\_index** (*int*) – The index for the edge
- **edge** (*object*) – The data payload/weight to update the edge with

**Raises** `NoEdgeBetweenNodes` – When there is no edge between nodes

**weighted\_edge\_list**()

Get edge list with weights

Returns a list of tuples of the form (`source`, `target`, `weight`) where `source` and `target` are the node indices and `weight` is the payload of the edge.

**Returns** An edge list with weights

**Return type** `WeightedEdgeList`

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## 2.1.2 networkx.PyDiGraph

**class** `PyDiGraph`(*check\_cycle=False, multigraph=True, /*)

A class for creating directed graphs

The `PyDiGraph` class is used to create a directed graph. It can be a multigraph (have multiple edges between nodes). Each node and edge (although rarely used for edges) is indexed by an integer id. Additionally each node and edge contains an arbitrary Python object as a weight/data payload. You can use the index for access to the data payload as in the following example:

```
import networkx

graph = networkx.PyDiGraph()
data_payload = "An arbitrary Python object"
node_index = graph.add_node(data_payload)
print("Node Index: %s" % node_index)
print(graph[node_index])
```

```
Node Index: 0
An arbitrary Python object
```

The `PyDiGraph` implements the Python mapping protocol for nodes so in addition to access you can also update the data payload with:

```
import networkx

graph = networkx.PyDiGraph()
data_payload = "An arbitrary Python object"
node_index = graph.add_node(data_payload)
graph[node_index] = "New Payload"
print("Node Index: %s" % node_index)
print(graph[node_index])
```

```
Node Index: 0
New Payload
```

The `PyDiGraph` class has an option for real time cycle checking which can be used to ensure any edges added to the graph does not introduce a cycle. By default the real time cycle checking feature is disabled for performance, however you can enable it by setting the `check_cycle` attribute to `True`. For example:

```
import networkx
dag = networkx.PyDiGraph()
dag.check_cycle = True
```

or at object creation:



```
import networkx
dag = networkx.PyDiGraph(check_cycle=True)
```

With `check_cycle` set to true any calls to `PyDiGraph.add_edge()` will ensure that no cycles are added, ensuring that the `PyDiGraph` class truly represents a directed acyclic graph. Do note that this cycle checking on `add_edge()`, `add_edges_from()`, `add_edges_from_no_data()`, `extend_from_edge_list()`, and `extend_from_weighted_edge_list()` comes with a performance penalty that grows as the graph does. If you're adding a node and edge at the same time leveraging `PyDiGraph.add_child()` or `PyDiGraph.add_parent()` will avoid this overhead.

`__init__()`

Initialize self. See `help(type(self))` for accurate signature.

## Methods

<code>__init__()</code>	Initialize self.
<code>add_child(parent, obj, edge, /)</code>	Add a new child node to the graph.
<code>add_edge(parent, child, edge, /)</code>	Add an edge between 2 nodes.
<code>add_edges_from(obj_list, /)</code>	Add new edges to the dag.
<code>add_edges_from_no_data(obj_list, /)</code>	Add new edges to the dag without python data.
<code>add_node(obj, /)</code>	Add a new node to the graph.
<code>add_nodes_from(obj_list, /)</code>	Add new nodes to the graph.
<code>add_parent(child, obj, edge, /)</code>	Add a new parent node to the dag.
<code>adj(node, /)</code>	Get the index and data for the neighbors of a node.
<code>adj_direction(node, direction, /)</code>	Get the index and data for either the parent or children of a node.
<code>compose(other, node_map, /[, node_map_func, ...])</code>	Add another <code>PyDiGraph</code> object into this <code>PyDiGraph</code>
<code>edge_list</code>	Get edge list
<code>edges()</code>	Return a list of all edge data.
<code>extend_from_edge_list(edge_list, /)</code>	Extend graph from an edge list
<code>extend_from_weighted_edge_list(edge_list, /)</code>	Extend graph from a weighted edge list
<code>find_adjacent_node_by_edge(node, predicate, /)</code>	Find a target node with a specific edge
<code>find_node_by_weight</code>	Find node within this graph given a specific weight
<code>from_adjacency_matrix(matrix, /)</code>	Create a new <code>PyDiGraph</code> object from an adjacency matrix
<code>get_all_edge_data(node_a, node_b, /)</code>	Return the edge data for all the edges between 2 nodes.
<code>get_edge_data(node_a, node_b, /)</code>	Return the edge data for an edge between 2 nodes.
<code>get_node_data(node, /)</code>	Return the node data for a given node index
<code>has_edge(node_a, node_b, /)</code>	Return True if there is an edge from node_a to node_b.
<code>in_degree(node, /)</code>	Get the degree of a node for inbound edges.
<code>in_edges(node, /)</code>	Get the index and edge data for all parents of a node.
<code>insert_node_on_in_edges(node, ref_node, /)</code>	Insert a node between a reference node and all its predecessor nodes
<code>insert_node_on_in_edges_multiple(node, ...)</code>	Insert a node between a list of reference nodes and all their predecessors
<code>insert_node_on_out_edges(node, ref_node, /)</code>	Insert a node between a reference node and all its successor nodes

continues on next page

Table 4 – continued from previous page

<code>insert_node_on_out_edges_multiple(node, ...)</code>	Insert a node between a list of reference nodes and all their successors
<code>is_symmetric()</code>	Check if the graph is symmetric
<code>merge_nodes(u, l, v)</code>	Merge two nodes in the graph.
<code>neighbors(node, l)</code>	Get the neighbors (i.e.
<code>node_indexes()</code>	Return a list of all node indexes.
<code>nodes()</code>	Return a list of all node data.
<code>out_degree(node, l)</code>	Get the degree of a node for outbound edges.
<code>out_edges(node, l)</code>	Get the index and edge data for all children of a node.
<code>predecessor_indices(node, l)</code>	Get the predecessor indices of a node.
<code>predecessors(node, l)</code>	Return a list of all the node predecessor data.
<code>read_edge_list(path, l[, comment, delimitator])</code>	Read an edge list file and create a new PyDiGraph object from the contents
<code>remove_edge(parent, child, l)</code>	Remove an edge between 2 nodes.
<code>remove_edge_from_index(edge, l)</code>	Remove an edge identified by the provided index
<code>remove_edges_from(index_list, l)</code>	Remove edges from the graph.
<code>remove_node(node, l)</code>	Remove a node from the graph.
<code>remove_node_retain_edges(node, l[, ...])</code>	Remove a node from the graph and add edges from all predecessors to all successors
<code>remove_nodes_from(index_list, l)</code>	Remove nodes from the graph.
<code>subgraph(nodes, l)</code>	Return a new PyDiGraph object for a subgraph of this graph
<code>successor_indices(node, l)</code>	Get the successor indices of a node.
<code>successors(node, l)</code>	Return a list of all the node successor data.
<code>to_dot([node_attr, edge_attr, graph_attr, ...])</code>	Generate a dot file from the graph
<code>to_undirected()</code>	Generate a new PyGraph object from this graph
<code>update_edge(source, target, l, edge)</code>	Update an edge's weight/payload inplace
<code>update_edge_by_index(source, target, l, edge)</code>	Update an edge's weight/payload by the edge index
<code>weighted_edge_list</code>	Get edge list with weights

### Attributes

<code>check_cycle</code>	Whether cycle checking is enabled for the Di-Graph/DAG.
<code>multigraph</code>	Whether the graph is a multigraph (allows multiple edges between nodes) or not

### `add_child(parent, obj, edge, l)`

Add a new child node to the graph.

This will create a new node on the graph and add an edge from the parent to that new node.

#### Parameters

- **parent** (*int*) – The index for the parent node
- **obj** – The python object to attach to the node
- **edge** – The python object to attach to the edge

**Returns** The index of the newly created child node

**Return type** int

**add\_edge**(*parent, child, edge, /*)

Add an edge between 2 nodes.

Use `add_child()` or `add_parent()` to create a node with an edge at the same time as an edge for better performance. Using this method will enable adding duplicate edges between nodes if the `check_cycle` attribute is set to `True`.

**Parameters**

- **parent** (*int*) – Index of the parent node
- **child** (*int*) – Index of the child node
- **edge** – The object to set as the data for the edge. It can be any python object.

**Returns** The edge index of the created edge

**Return type** `int`

**Raises** When the new edge will create a cycle

**add\_edges\_from**(*obj\_list, /*)

Add new edges to the dag.

**Parameters** **obj\_list** (*list*) – A list of tuples of the form (`parent`, `child`, `obj`) to attach to the graph. `parent` and `child` are integer indexes describing where an edge should be added, and `obj` is the python object for the edge data.

**Returns** A list of int indices of the newly created edges

**Return type** `list`

**add\_edges\_from\_no\_data**(*obj\_list, /*)

Add new edges to the dag without python data.

**Parameters** **obj\_list** (*list*) – A list of tuples of the form (`parent`, `child`) to attach to the graph. `parent` and `child` are integer indexes describing where an edge should be added. Unlike `add_edges_from()` there is no data payload and when the edge is created `None` will be used.

**Returns** A list of int indices of the newly created edges

**Return type** `list`

**add\_node**(*obj, /*)

Add a new node to the graph.

**Parameters** **obj** – The python object to attach to the node

**Returns** The index of the newly created node

**Return type** `int`

**add\_nodes\_from**(*obj\_list, /*)

Add new nodes to the graph.

**Parameters** **obj\_list** (*list*) – A list of python objects to attach to the graph as new nodes

**Returns** A list of int indices of the newly created nodes

**Return type** `NodeIndices`

**add\_parent**(*child, obj, edge, /*)

Add a new parent node to the dag.

This create a new node on the dag and add an edge to the child from that new node

**Parameters**

- **child** (*int*) – The index of the child node
- **obj** – The python object to attach to the node
- **edge** – The python object to attach to the edge

**Returns** **index** The index of the newly created parent node

**Return type** `int`

**adj**(*node, /*)

Get the index and data for the neighbors of a node.

This will return a dictionary where the keys are the node indexes of the adjacent nodes (inbound or outbound) and the value is the edge data objects between that adjacent node and the provided node. Note in the case of a multigraph only one edge will be used, not all of the edges between two node.

**Parameters** **node** (*int*) – The index of the node to get the neighbors

**Returns** A dictionary where the keys are node indexes and the value is the edge data object for all nodes that share an edge with the specified node.

**Return type** `dict`

**adj\_direction**(*node, direction, /*)

Get the index and data for either the parent or children of a node.

This will return a dictionary where the keys are the node indexes of the adjacent nodes (inbound or outbound as specified) and the value is the edge data objects for the edges between that adjacent node and the provided node. Note in the case of a multigraph only one edge one edge will be used, not all of the edges between two node.

**Parameters**

- **node** (*int*) – The index of the node to get the neighbors
- **direction** (*bool*) – The direction to use for finding nodes, True means inbound edges and False means outbound edges.

**Returns** A dictionary where the keys are node indexes and the value is the edge data object for all nodes that share an edge with the specified node.

**Return type** `dict`

**check\_cycle**

Whether cycle checking is enabled for the DiGraph/DAG.

If set to True adding new edges that would introduce a cycle will raise a [\*DAGWouldCycle\*](#) exception.

**compose**(*other, node\_map, /, node\_map\_func=None, edge\_map\_func=None*)

Add another PyDiGraph object into this PyDiGraph

**Parameters**

- **other** ([PyDiGraph](#)) – The other PyDiGraph object to add onto this graph.
- **node\_map** (*dict*) – A dictionary mapping node indexes from this PyDiGraph object to node indexes in the other PyDiGraph object. The keys are a node index in this graph and the value is a tuple of the node index in the other graph to add an edge to and the weight of that edge. For example:

```
{
  1: (2, "weight"),
  2: (4, "weight2")
}
```

- **node\_map\_func** – An optional python callable that will take in a single node weight/data object and return a new node weight/data object that will be used when adding an node from other onto this graph.
- **edge\_map\_func** – An optional python callable that will take in a single edge weight/data object and return a new edge weight/data object that will be used when adding an edge from other onto this graph.

**Returns** new\_node\_ids: A dictionary mapping node index from the other PyDiGraph to the corresponding node index in this PyDAG after they've been combined

**Return type** dict

For example, start by building a graph:

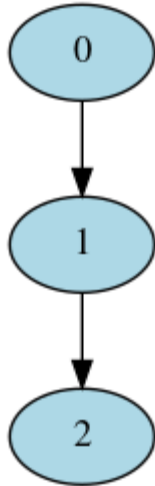
```
import os
import tempfile

import pydot
from PIL import Image

import networkx

# Build first graph and visualize:
graph = networkx.PyDiGraph()
node_a = graph.add_node('A')
node_b = graph.add_child(node_a, 'B', 'A to B')
node_c = graph.add_child(node_b, 'C', 'B to C')
dot_str = graph.to_dot(
    lambda node: dict(
        color='black', fillcolor='lightblue', style='filled')
)
dot = pydot.graph_from_dot_data(dot_str)[0]

with tempfile.TemporaryDirectory() as tmpdirname:
    tmp_path = os.path.join(tmpdirname, 'graph.png')
    dot.write_png(tmp_path)
    image = Image.open(tmp_path)
    os.remove(tmp_path)
image
```



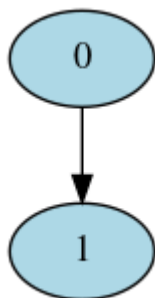
Then build a second one:

```

# Build second graph and visualize:
other_graph = networkx.PyDiGraph()
node_d = other_graph.add_node('D')
other_graph.add_child(node_d, 'E', 'D to E')
dot_str = other_graph.to_dot(
    lambda node: dict(
        color='black', fillcolor='lightblue', style='filled'))
dot = pydot.graph_from_dot_data(dot_str)[0]

with tempfile.TemporaryDirectory() as tmpdirname:
    tmp_path = os.path.join(tmpdirname, 'other_graph.png')
    dot.write_png(tmp_path)
    image = Image.open(tmp_path)
    os.remove(tmp_path)
image

```



Finally compose the other\_graph onto graph

```

node_map = {node_b: (node_d, 'B to D')}
graph.compose(other_graph, node_map)
dot_str = graph.to_dot(
    lambda node: dict(
        color='black', fillcolor='lightblue', style='filled'))
dot = pydot.graph_from_dot_data(dot_str)[0]

```

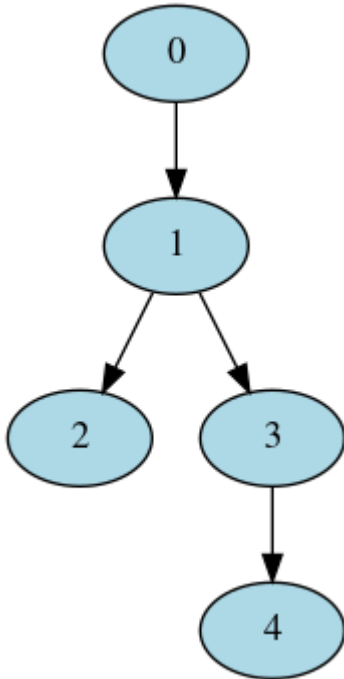
(continues on next page)

(continued from previous page)

```

with tempfile.TemporaryDirectory() as tmpdirname:
    tmp_path = os.path.join(tmpdirname, 'combined_graph.png')
    dot.write_png(tmp_path)
    image = Image.open(tmp_path)
    os.remove(tmp_path)
image

```

**edge\_list()**

Get edge list

Returns a list of tuples of the form (source, target) where source and target are the node indices.

**Returns** An edge list with weights**Return type** *EdgeList***edges()**

Return a list of all edge data.

**Returns** A list of all the edge data objects in the graph**Return type** list**extend\_from\_edge\_list(edge\_list, /)**

Extend graph from an edge list

This method differs from [add\\_edges\\_from\\_no\\_data\(\)](#) in that it will add nodes if a node index is not present in the edge list.**Parameters** **edge\_list** (*list*) – A list of tuples of the form (source, target) where source and target are integer node indices. If the node index is not present in the graph, nodes will be added (with a node weight of None) to that index.**extend\_from\_weighted\_edge\_list(edge\_list, /)**

Extend graph from a weighted edge list

This method differs from `add_edges_from()` in that it will add nodes if a node index is not present in the edge list.

**Parameters** `edge_list` (*list*) – A list of tuples of the form (source, target, weight) where source and target are integer node indices. If the node index is not present in the graph nodes will be added (with a node weight of `None`) to that index.

**find\_adjacent\_node\_by\_edge**(*node, predicate, /*)

Find a target node with a specific edge

This method is used to find a target node that is adjacent to a given node given an edge condition.

**Parameters**

- **node** (*int*) – The node to use as the source of the search
- **predicate** (*callable*) – A python callable that will take a single parameter, the edge object, and will return a boolean if the edge matches or not

**Returns** The node object that has an edge to it from the provided node index which matches the provided condition

**find\_node\_by\_weight**(*/*)

Find node within this graph given a specific weight

This algorithm has a worst case of  $O(n)$  since it searches the node indices in order. If there is more than one node in the graph with the same weight only the first match (by node index) will be returned.

**Parameters** `obj` – The weight to look for in the graph.

**Returns** the index of the first node in the graph that is equal to the weight. If no match is found `None` will be returned.

**Return type** `int`

**static from\_adjacency\_matrix**(*matrix, /*)

Create a new `PyDiGraph` object from an adjacency matrix

This method can be used to construct a new `PyDiGraph` object from an input adjacency matrix. The node weights will be the index from the matrix. The edge weights will be a float value of the value from the matrix.

**Parameters** `matrix` (*ndarray*) – The input numpy array adjacency matrix to create a new `PyDiGraph` object from. It must be a 2 dimensional array and be a `float/np.float64` data type.

**Returns** A new graph object generated from the adjacency matrix

**Return type** `PyDiGraph`

**get\_all\_edge\_data**(*node\_a, node\_b, /*)

Return the edge data for all the edges between 2 nodes.

**Parameters**

- **node\_a** (*int*) – The index for the first node
- **node\_b** (*int*) – The index for the second node

**Returns** A list with all the data objects for the edges between nodes

**Return type** `list`

**Raises** `NoEdgeBetweenNodes` – When there is no edge between nodes



**get\_edge\_data**(*node\_a*, *node\_b*, /)

Return the edge data for an edge between 2 nodes.

**Parameters**

- **node\_a** (*int*) – The index for the first node
- **node\_b** (*int*) – The index for the second node

**Returns** The data object set for the edge

**Raises** *NoEdgeBetweenNodes* – When there is no edge between nodes

**get\_node\_data**(*node*, /)

Return the node data for a given node index

**Parameters** **node** (*int*) – The index for the node

**Returns** The data object set for that node

**Raises** *IndexError* – when an invalid node index is provided

**has\_edge**(*node\_a*, *node\_b*, /)

Return True if there is an edge from *node\_a* to *node\_b*.

**Parameters**

- **node\_a** (*int*) – The source node index to check for an edge
- **node\_b** (*int*) – The destination node index to check for an edge

**Returns** True if there is an edge false if there is no edge

**Return type** bool

**in\_degree**(*node*, /)

Get the degree of a node for inbound edges.

**Parameters** **node** (*int*) – The index of the node to find the inbound degree of

**Returns** The inbound degree for the specified node

**Return type** int

**in\_edges**(*node*, /)

Get the index and edge data for all parents of a node.

This will return a list of tuples with the parent index the node index and the edge data. This can be used to recreate `add_edge()` calls. `node`: The index of the node to get the edges for

**Parameters** **node** (*int*) – The index of the node to get the edges for

**Returns** A list of tuples of the form: (parent\_index, node\_index, edge\_data)

**Return type** *WeightedEdgeList*

**insert\_node\_on\_in\_edges**(*node*, *ref\_node*, /)

Insert a node between a reference node and all its predecessor nodes

This essentially iterates over all edges into the reference node specified in the `ref_node` parameter removes those edges and then adds 2 edges, one from the predecessor of `ref_node` to `node` and the other from `node` to `ref_node`. The edge payloads for the newly created edges are copied by reference from the original edge that gets removed.

**Parameters**

- **node** (*int*) – The node index to insert between

- **ref\_node** (*int*) – The reference node index to insert node between

**insert\_node\_on\_in\_edges\_multiple**(*node, ref\_nodes, /*)

Insert a node between a list of reference nodes and all their predecessors

This essentially iterates over all edges into the reference node specified in the **ref\_nodes** parameter removes those edges and then adds 2 edges, one from the predecessor of **ref\_node** to **node** and the other from **node** to **ref\_node**. The edge payloads for the newly created edges are copied by reference from the original edge that gets removed.

#### Parameters

- **node** (*int*) – The node index to insert between
- **ref\_node** (*int*) – The reference node index to insert node between

**insert\_node\_on\_out\_edges**(*node, ref\_node, /*)

Insert a node between a reference node and all its successor nodes

This essentially iterates over all edges out of the reference node specified in the **ref\_node** parameter removes those edges and then adds 2 edges, one from **ref\_node** to **node** and the other from **node** to the successor of **ref\_node**. The edge payloads for the newly created edges are copied by reference from the original edge that gets removed.

#### Parameters

- **node** (*int*) – The node index to insert between
- **ref\_node** (*int*) – The reference node index to insert node between

**insert\_node\_on\_out\_edges\_multiple**(*node, ref\_nodes, /*)

Insert a node between a list of reference nodes and all their successors

This essentially iterates over all edges out of the reference node specified in the **ref\_node** parameter removes those edges and then adds 2 edges, one from **ref\_node** to **node** and the other from **node** to the successor of **ref\_node**. The edge payloads for the newly created edges are copied by reference from the original edge that gets removed.

#### Parameters

- **node** (*int*) – The node index to insert between
- **ref\_nodes** (*int*) – The list of node indices to insert node between

**is\_symmetric**()

Check if the graph is symmetric

**Returns** True if the graph is symmetric

**Return type** bool

**merge\_nodes**(*u, /, v*)

Merge two nodes in the graph.

If the nodes have equal weight objects then all the edges into and out of *u* will be added to *v* and *u* will be removed from the graph. If the nodes don't have equal weight objects then no changes will be made and no error raised

#### Parameters

- **u** (*int*) – The source node that is going to be merged
- **v** (*int*) – The target node that is going to be the new node

**multigraph**

Whether the graph is a multigraph (allows multiple edges between nodes) or not

If set to `False` multiple edges between nodes are not allowed and calls that would add a parallel edge will instead update the existing edge

**neighbors**(*node*, /)

Get the neighbors (i.e. successors) of a node.

This will return a list of neighbor node indices. This function is equivalent to `successor_indices()`.

**Parameters** **node** (*int*) – The index of the node to get the neighbors of

**Returns** A list of the neighbor node indices

**Return type** *NodeIndices*

**node\_indexes**()

Return a list of all node indexes.

**Returns** A list of all the node indexes in the graph

**Return type** *NodeIndices*

**nodes**()

Return a list of all node data.

**Returns** A list of all the node data objects in the graph

**Return type** list

**out\_degree**(*node*, /)

Get the degree of a node for outbound edges.

**Parameters** **node** (*int*) – The index of the node to find the outbound degree of

**Returns** The outbound degree for the specified node

**Return type** int

**out\_edges**(*node*, /)

Get the index and edge data for all children of a node.

This will return a list of tuples with the child index the node index and the edge data. This can be used to recreate `add_edge()` calls.

**Parameters** **node** (*int*) – The index of the node to get the edges for

**Returns** **out\_edges** A list of tuples of the form: ``(node_index, child_index, edge_data)``

**Return type** *WeightedEdgeList*

**predecessor\_indices**(*node*, /)

Get the predecessor indices of a node.

This will return a list of the node indices for the predecessors of a node

**Parameters** **node** (*int*) – The index of the node to get the predecessors of

**Returns** A list of the neighbor node indices

**Return type** *NodeIndices*

**predecessors**(*node*, /)

Return a list of all the node predecessor data.

**Parameters** **node** (*int*) – The index for the node to get the predecessors for

**Returns** A list of the node data for all the parent neighbor nodes

**Return type** list

**static read\_edge\_list**(*path*, /, *comment=None*, *delimiter=None*)

Read an edge list file and create a new PyDiGraph object from the contents

The expected format for the edge list file is a line separated list of delimited node ids. If there are more than 3 elements on a line the 3rd on will be treated as a string weight for the edge

**Parameters**

- **path** (*str*) – The path of the file to open
- **comment** (*str*) – Optional character to use as a comment by default there are no comment characters
- **delimiter** (*str*) – Optional character to use as a delimiter by default any whitespace will be used

For example:

```
import os
import tempfile

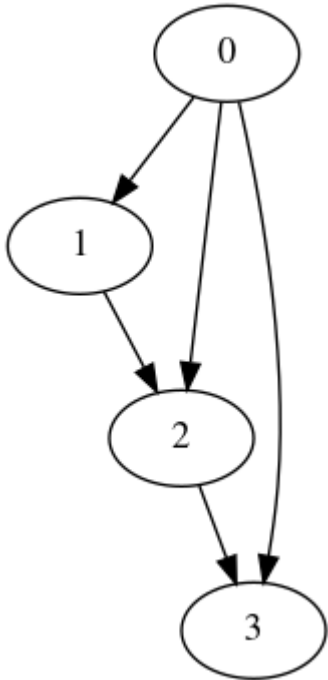
from PIL import Image
import pydot

import networkx

with tempfile.NamedTemporaryFile('wt') as fd:
    path = fd.name
    fd.write('0 1\n')
    fd.write('0 2\n')
    fd.write('0 3\n')
    fd.write('1 2\n')
    fd.write('2 3\n')
    fd.flush()
    graph = networkx.PyDiGraph.read_edge_list(path)

# Draw graph
dot = pydot.graph_from_dot_data(graph.to_dot())[0]

with tempfile.TemporaryDirectory() as tmpdirname:
    tmp_path = os.path.join(tmpdirname, 'dag.png')
    dot.write_png(tmp_path)
    image = Image.open(tmp_path)
    os.remove(tmp_path)
image
```



**remove\_edge**(parent, child, /)

Remove an edge between 2 nodes.

Note if there are multiple edges between the specified nodes only one will be removed.

**Parameters**

- **parent** (*int*) – The index for the parent node.
- **child** (*int*) – The index of the child node.

**Raises** *NoEdgeBetweenNodes* – If there are no edges between the nodes specified

**remove\_edge\_from\_index**(edge, /)

Remove an edge identified by the provided index

**Parameters** **edge** (*int*) – The index of the edge to remove

**remove\_edges\_from**(index\_list, /)

Remove edges from the graph.

Note if there are multiple edges between the specified nodes only one will be removed.

**Parameters** **index\_list** (*list*) – A list of node index pairs to remove from the graph

**remove\_node**(node, /)

Remove a node from the graph.

**Parameters** **node** (*int*) – The index of the node to remove. If the index is not present in the graph it will be ignored and this function will have no effect.

**remove\_node\_retain\_edges**(node, /, use\_outgoing=None, condition=None)

Remove a node from the graph and add edges from all predecessors to all successors

By default the data/weight on edges into the removed node will be used for the retained edges.

**Parameters**

- **node** (*int*) – The index of the node to remove. If the index is not present in the graph it will be ignored and this function will have no effect.
- **use\_outgoing** (*bool*) – If set to true the weight/data from the edge outgoing from node will be used in the retained edge instead of the default weight/data from the incoming edge.
- **condition** – A callable that will be passed 2 edge weight/data objects, one from the incoming edge to node the other for the outgoing edge, and will return a bool on whether an edge should be retained. For example setting this kwarg to:

```
lambda in_edge, out_edge: in_edge == out_edge
```

would only retain edges if the input edge to node had the same data payload as the outgoing edge.

**remove\_nodes\_from**(*index\_list, /*)

Remove nodes from the graph.

If a node index in the list is not present in the graph it will be ignored.

**Parameters** **index\_list** (*list*) – A list of node indices to remove from the the graph.

**subgraph**(*nodes, /*)

Return a new PyDiGraph object for a subgraph of this graph

**Parameters** **nodes** (*list*) – A list of node indices to generate the subgraph from. If a node index is included that is not present in the graph it will silently be ignored.

**Returns** A new PyDiGraph object representing a subgraph of this graph. It is worth noting that node and edge weight/data payloads are passed by reference so if you update (not replace) an object used as the weight in graph or the subgraph it will also be updated in the other.

**Return type** *PyGraph*

**successor\_indices**(*node, /*)

Get the successor indices of a node.

This will return a list of the node indices for the successors of a node

**Parameters** **node** (*int*) – The index of the node to get the successors of

**Returns** A list of the neighbor node indices

**Return type** *NodeIndices*

**successors**(*node, /*)

Return a list of all the node successor data.

**Parameters** **node** (*int*) – The index for the node to get the successors for

**Returns** A list of the node data for all the child neighbor nodes

**Return type** list

**to\_dot**(*node\_attr=None, edge\_attr=None, graph\_attr=None, filename=None*)

Generate a dot file from the graph

**Parameters**

- **node\_attr** – A callable that will take in a node data object and return a dictionary of attributes to be associated with the node in the dot file. The key and value of this dictionary **must** be strings. If they're not strings networkx will raise TypeError (unfortunately without an error message because of current limitations in the PyO3 type checking)

- **edge\_attr** – A callable that will take in an edge data object and return a dictionary of attributes to be associated with the node in the dot file. The key and value of this dictionary **must** be a string. If they're not strings networkx will raise TypeError (unfortunately without an error message because of current limitations in the PyO3 type checking)
- **graph\_attr** (*dict*) – An optional dictionary that specifies any graph attributes for the output dot file. The key and value of this dictionary **must** be a string. If they're not strings networkx will raise TypeError (unfortunately without an error message because of current limitations in the PyO3 type checking)
- **filename** (*str*) – An optional path to write the dot file to if specified there is no return from the function

**Returns** A string with the dot file contents if filename is not specified.

**Return type** str

Using this method enables you to leverage graphviz to visualize a [networkx.PyDiGraph](#) object. For example:

```
import os
import tempfile

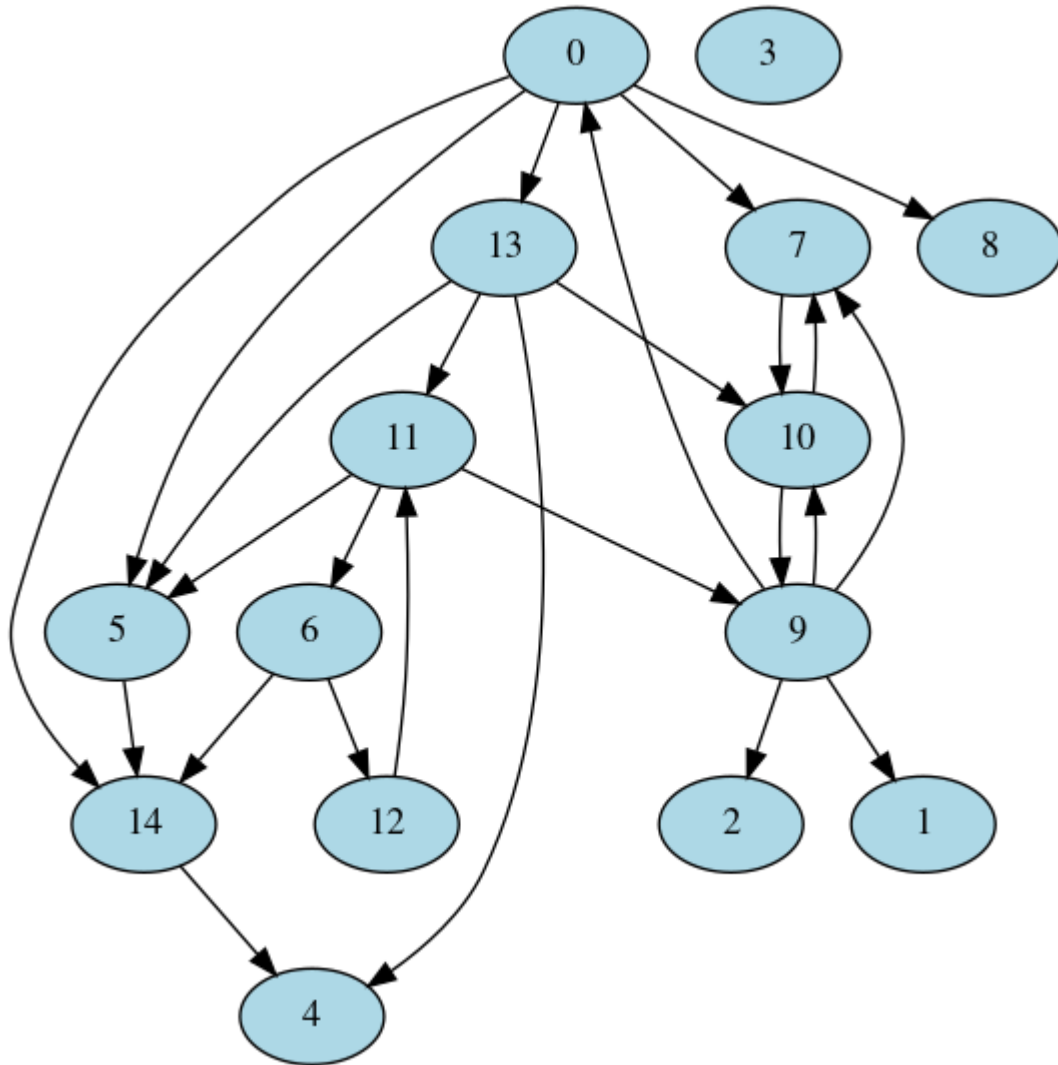
import pydot
from PIL import Image

import networkx

graph = networkx.directed_gnp_random_graph(15, .25)
dot_str = graph.to_dot(
    lambda node: dict(
        color='black', fillcolor='lightblue', style='filled'))
dot = pydot.graph_from_dot_data(dot_str)[0]

with tempfile.TemporaryDirectory() as tmpdirname:
    tmp_path = os.path.join(tmpdirname, 'dag.png')
    dot.write_png(tmp_path)
    image = Image.open(tmp_path)
    os.remove(tmp_path)

image
```

**to\_undirected()**

Generate a new PyGraph object from this graph

This will create a new *PyGraph* object from this graph. All edges in this graph will be created as undirected edges in the new graph object. Do note that the node and edge weights/data payloads will be passed by reference to the new *PyGraph* object.

**Returns** A new PyGraph object with an undirected edge for every directed edge in this graph

**Return type** *PyGraph*

**update\_edge(source, target, /, edge)**

Update an edge's weight/payload inplace

If there are parallel edges in the graph only one edge will be updated. if you need to update a specific edge or need to ensure all parallel edges get updated you should use *update\_edge\_by\_index()* instead.

**Parameters**

- **source** (*int*) – The index for the first node
- **target** (*int*) – The index for the second node

**Raises** *NoEdgeBetweenNodes* – When there is no edge between nodes



**update\_edge\_by\_index**(*source, target, /, edge*)

Update an edge's weight/payload by the edge index

**Parameters**

- **edge\_index** (*int*) – The index for the edge
- **edge** (*object*) – The data payload/weight to update the edge with

**Raises** *NoEdgeBetweenNodes* – When there is no edge between nodes

**weighted\_edge\_list**()

Get edge list with weights

Returns a list of tuples of the form (*source, target, weight*) where *source* and *target* are the node indices and *weight* is the payload of the edge.

**Returns** An edge list with weights

**Return type** *WeightedEdgeList*

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

### 2.1.3 networkx.PyDAG

**class PyDAG**(*check\_cycle=False, multigraph=True, /*)

A class for creating direct acyclic graphs.

PyDAG is just an alias of the PyDiGraph class and behaves identically to the *PyDiGraph* class and can be used interchangeably with PyDiGraph. It currently exists solely as a backwards compatibility alias for users of networkx from prior to the 0.4.0 release when there was no PyDiGraph class.

The PyDAG class is used to create a directed graph. It can be a multigraph (have multiple edges between nodes). Each node and edge (although rarely used for edges) is indexed by an integer id. Additionally, each node and edge contains an arbitrary Python object as a weight/data payload.

You can use the index for access to the data payload as in the following example:

```
import networkx

graph = networkx.PyDAG()
data_payload = "An arbitrary Python object"
node_index = graph.add_node(data_payload)
print("Node Index: %s" % node_index)
print(graph[node_index])
```

```
Node Index: 0
An arbitrary Python object
```

The PyDAG class implements the Python mapping protocol for nodes so in addition to access you can also update the data payload with:

```
import networkx

graph = networkx.PyDAG()
data_payload = "An arbitrary Python object"
node_index = graph.add_node(data_payload)
graph[node_index] = "New Payload"
print("Node Index: %s" % node_index)
print(graph[node_index])
```

```
Node Index: 0
New Payload
```

The PyDAG class has an option for real time cycle checking which can be used to ensure any edges added to the graph does not introduce a cycle. By default the real time cycle checking feature is disabled for performance, however you can enable it by setting the `check_cycle` attribute to `True`. For example:

```
import networkx
dag = networkx.PyDAG()
dag.check_cycle = True
```

or at object creation:

```
import networkx
dag = networkx.PyDAG(check_cycle=True)
```

With `check_cycle` set to `true` any calls to `PyDAG.add_edge()` will ensure that no cycles are added, ensuring that the PyDAG class truly represents a directed acyclic graph. Do note that this cycle checking on `add_edge()`, `add_edges_from()`, `add_edges_from_no_data()`, `extend_from_edge_list()`, and `extend_from_weighted_edge_list()` comes with a performance penalty that grows as the graph does. If you're adding a node and edge at the same time, leveraging `PyDAG.add_child()` or `PyDAG.add_parent()` will avoid this overhead.

```
__init__()
    Initialize self. See help(type(self)) for accurate signature.
```

## Methods

<code>__init__()</code>	Initialize self.
<code>add_child(parent, obj, edge, /)</code>	Add a new child node to the graph.
<code>add_edge(parent, child, edge, /)</code>	Add an edge between 2 nodes.
<code>add_edges_from(obj_list, /)</code>	Add new edges to the dag.
<code>add_edges_from_no_data(obj_list, /)</code>	Add new edges to the dag without python data.
<code>add_node(obj, /)</code>	Add a new node to the graph.
<code>add_nodes_from(obj_list, /)</code>	Add new nodes to the graph.
<code>add_parent(child, obj, edge, /)</code>	Add a new parent node to the dag.
<code>adj(node, /)</code>	Get the index and data for the neighbors of a node.
<code>adj_direction(node, direction, /)</code>	Get the index and data for either the parent or children of a node.
<code>compose(other, node_map, /[, node_map_func, ...])</code>	Add another PyDiGraph object into this PyDiGraph
<code>edge_list</code>	Get edge list
<code>edges()</code>	Return a list of all edge data.

continues on next page

Table 6 – continued from previous page

<code>extend_from_edge_list(edge_list, /)</code>	Extend graph from an edge list
<code>extend_from_weighted_edge_list(edge_list, /)</code>	Extend graph from a weighted edge list
<code>find_adjacent_node_by_edge(node, predicate, /)</code>	Find a target node with a specific edge
<code>find_node_by_weight</code>	Find node within this graph given a specific weight
<code>from_adjacency_matrix(matrix, /)</code>	Create a new <code>PyDiGraph</code> object from an adjacency matrix
<code>get_all_edge_data(node_a, node_b, /)</code>	Return the edge data for all the edges between 2 nodes.
<code>get_edge_data(node_a, node_b, /)</code>	Return the edge data for an edge between 2 nodes.
<code>get_node_data(node, /)</code>	Return the node data for a given node index
<code>has_edge(node_a, node_b, /)</code>	Return True if there is an edge from node_a to node_b.
<code>in_degree(node, /)</code>	Get the degree of a node for inbound edges.
<code>in_edges(node, /)</code>	Get the index and edge data for all parents of a node.
<code>insert_node_on_in_edges(node, ref_node, /)</code>	Insert a node between a reference node and all its predecessor nodes
<code>insert_node_on_in_edges_multiple(node, ...)</code>	Insert a node between a list of reference nodes and all their predecessors
<code>insert_node_on_out_edges(node, ref_node, /)</code>	Insert a node between a reference node and all its successor nodes
<code>insert_node_on_out_edges_multiple(node, ...)</code>	Insert a node between a list of reference nodes and all their successors
<code>is_symmetric()</code>	Check if the graph is symmetric
<code>merge_nodes(u, /, v)</code>	Merge two nodes in the graph.
<code>neighbors(node, /)</code>	Get the neighbors (i.e.
<code>node_indexes()</code>	Return a list of all node indexes.
<code>nodes()</code>	Return a list of all node data.
<code>out_degree(node, /)</code>	Get the degree of a node for outbound edges.
<code>out_edges(node, /)</code>	Get the index and edge data for all children of a node.
<code>predecessor_indices(node, /)</code>	Get the predecessor indices of a node.
<code>predecessors(node, /)</code>	Return a list of all the node predecessor data.
<code>read_edge_list(path, /[, comment, delimitator])</code>	Read an edge list file and create a new <code>PyDiGraph</code> object from the contents
<code>remove_edge(parent, child, /)</code>	Remove an edge between 2 nodes.
<code>remove_edge_from_index(edge, /)</code>	Remove an edge identified by the provided index
<code>remove_edges_from(index_list, /)</code>	Remove edges from the graph.
<code>remove_node(node, /)</code>	Remove a node from the graph.
<code>remove_node_retain_edges(node, /[, ...])</code>	Remove a node from the graph and add edges from all predecessors to all successors
<code>remove_nodes_from(index_list, /)</code>	Remove nodes from the graph.
<code>subgraph(nodes, /)</code>	Return a new <code>PyDiGraph</code> object for a subgraph of this graph
<code>successor_indices(node, /)</code>	Get the successor indices of a node.
<code>successors(node, /)</code>	Return a list of all the node successor data.
<code>to_dot([node_attr, edge_attr, graph_attr, ...])</code>	Generate a dot file from the graph
<code>to_undirected()</code>	Generate a new <code>PyGraph</code> object from this graph
<code>update_edge(source, target, /, edge)</code>	Update an edge's weight/payload inplace
<code>update_edge_by_index(source, target, /, edge)</code>	Update an edge's weight/payload by the edge index
<code>weighted_edge_list</code>	Get edge list with weights

## Attributes

<code>check_cycle</code>	Whether cycle checking is enabled for the Di-Graph/DAG.
<code>multigraph</code>	Whether the graph is a multigraph (allows multiple edges between nodes) or not

**add\_child**(*parent, obj, edge, /*)

Add a new child node to the graph.

This will create a new node on the graph and add an edge from the parent to that new node.

### Parameters

- **parent** (*int*) – The index for the parent node
- **obj** – The python object to attach to the node
- **edge** – The python object to attach to the edge

**Returns** The index of the newly created child node

**Return type** int

**add\_edge**(*parent, child, edge, /*)

Add an edge between 2 nodes.

Use `add_child()` or `add_parent()` to create a node with an edge at the same time as an edge for better performance. Using this method will enable adding duplicate edges between nodes if the `check_cycle` attribute is set to `True`.

### Parameters

- **parent** (*int*) – Index of the parent node
- **child** (*int*) – Index of the child node
- **edge** – The object to set as the data for the edge. It can be any python object.

**Returns** The edge index of the created edge

**Return type** int

**Raises** When the new edge will create a cycle

**add\_edges\_from**(*obj\_list, /*)

Add new edges to the dag.

**Parameters** **obj\_list** (*list*) – A list of tuples of the form (`parent`, `child`, `obj`) to attach to the graph. `parent` and `child` are integer indexes describing where an edge should be added, and `obj` is the python object for the edge data.

**Returns** A list of int indices of the newly created edges

**Return type** list

**add\_edges\_from\_no\_data**(*obj\_list, /*)

Add new edges to the dag without python data.

**Parameters** **obj\_list** (*list*) – A list of tuples of the form (`parent`, `child`) to attach to the graph. `parent` and `child` are integer indexes describing where an edge should be added. Unlike `add_edges_from()` there is no data payload and when the edge is created `None` will be used.

**Returns** A list of int indices of the newly created edges

**Return type** list

**add\_node**(*obj*, /)

Add a new node to the graph.

**Parameters** **obj** – The python object to attach to the node

**Returns** The index of the newly created node

**Return type** int

**add\_nodes\_from**(*obj\_list*, /)

Add new nodes to the graph.

**Parameters** **obj\_list** (*list*) – A list of python objects to attach to the graph as new nodes

**Returns** A list of int indices of the newly created nodes

**Return type** *NodeIndices*

**add\_parent**(*child*, *obj*, *edge*, /)

Add a new parent node to the dag.

This create a new node on the dag and add an edge to the child from that new node

**Parameters**

- **child** (*int*) – The index of the child node
- **obj** – The python object to attach to the node
- **edge** – The python object to attach to the edge

**Returns** **index** The index of the newly created parent node

**Return type** int

**adj**(*node*, /)

Get the index and data for the neighbors of a node.

This will return a dictionary where the keys are the node indexes of the adjacent nodes (inbound or outbound) and the value is the edge data objects between that adjacent node and the provided node. Note in the case of a multigraph only one edge will be used, not all of the edges between two node.

**Parameters** **node** (*int*) – The index of the node to get the neighbors

**Returns** A dictionary where the keys are node indexes and the value is the edge data object for all nodes that share an edge with the specified node.

**Return type** dict

**adj\_direction**(*node*, *direction*, /)

Get the index and data for either the parent or children of a node.

This will return a dictionary where the keys are the node indexes of the adjacent nodes (inbound or outbound as specified) and the value is the edge data objects for the edges between that adjacent node and the provided node. Note in the case of a multigraph only one edge one edge will be used, not all of the edges between two node.

**Parameters**

- **node** (*int*) – The index of the node to get the neighbors
- **direction** (*bool*) – The direction to use for finding nodes, True means inbound edges and False means outbound edges.

**Returns** A dictionary where the keys are node indexes and the value is the edge data object for all nodes that share an edge with the specified node.

**Return type** dict

### check\_cycle

Whether cycle checking is enabled for the DiGraph/DAG.

If set to True adding new edges that would introduce a cycle will raise a `DAGWouldCycle` exception.

**compose**(*other*, *node\_map*, /, *node\_map\_func*=None, *edge\_map\_func*=None)

Add another PyDiGraph object into this PyDiGraph

### Parameters

- **other** (`PyDiGraph`) – The other PyDiGraph object to add onto this graph.
- **node\_map** (`dict`) – A dictionary mapping node indexes from this PyDiGraph object to node indexes in the other PyDiGraph object. The keys are a node index in this graph and the value is a tuple of the node index in the other graph to add an edge to and the weight of that edge. For example:

```
{
  1: (2, "weight"),
  2: (4, "weight2")
}
```

- **node\_map\_func** – An optional python callable that will take in a single node weight/data object and return a new node weight/data object that will be used when adding a node from other onto this graph.
- **edge\_map\_func** – An optional python callable that will take in a single edge weight/data object and return a new edge weight/data object that will be used when adding an edge from other onto this graph.

**Returns** `new_node_ids`: A dictionary mapping node index from the other PyDiGraph to the corresponding node index in this PyDAG after they've been combined

**Return type** dict

For example, start by building a graph:

```
import os
import tempfile

import pydot
from PIL import Image

import networkx

# Build first graph and visualize:
graph = networkx.PyDiGraph()
node_a = graph.add_node('A')
node_b = graph.add_child(node_a, 'B', 'A to B')
node_c = graph.add_child(node_b, 'C', 'B to C')
dot_str = graph.to_dot(
    lambda node: dict(
        color='black', fillcolor='lightblue', style='filled'))
dot = pydot.graph_from_dot_data(dot_str)[0]
```

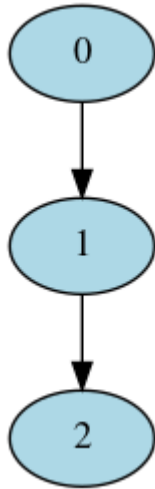
(continues on next page)

(continued from previous page)

```

with tempfile.TemporaryDirectory() as tmpdirname:
    tmp_path = os.path.join(tmpdirname, 'graph.png')
    dot.write_png(tmp_path)
    image = Image.open(tmp_path)
    os.remove(tmp_path)
image

```



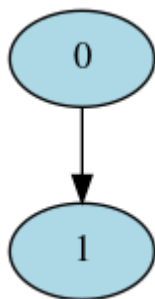
Then build a second one:

```

# Build second graph and visualize:
other_graph = networkx.PyDiGraph()
node_d = other_graph.add_node('D')
other_graph.add_child(node_d, 'E', 'D to E')
dot_str = other_graph.to_dot(
    lambda node: dict(
        color='black', fillcolor='lightblue', style='filled'))
dot = pydot.graph_from_dot_data(dot_str)[0]

with tempfile.TemporaryDirectory() as tmpdirname:
    tmp_path = os.path.join(tmpdirname, 'other_graph.png')
    dot.write_png(tmp_path)
    image = Image.open(tmp_path)
    os.remove(tmp_path)
image

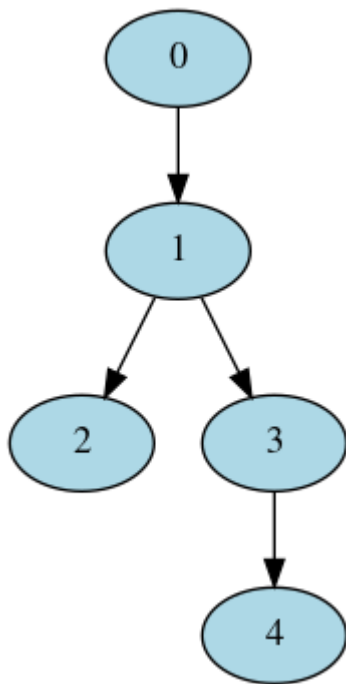
```



Finally compose the `other_graph` onto `graph`

```
node_map = {node_b: (node_d, 'B to D')}
graph.compose(other_graph, node_map)
dot_str = graph.to_dot(
    lambda node: dict(
        color='black', fillcolor='lightblue', style='filled'))
dot = pydot.graph_from_dot_data(dot_str)[0]

with tempfile.TemporaryDirectory() as tmpdirname:
    tmp_path = os.path.join(tmpdirname, 'combined_graph.png')
    dot.write_png(tmp_path)
    image = Image.open(tmp_path)
    os.remove(tmp_path)
image
```



### `edge_list()`

Get edge list

Returns a list of tuples of the form (source, target) where source and target are the node indices.

**Returns** An edge list with weights

**Return type** *EdgeList*

### `edges()`

Return a list of all edge data.

**Returns** A list of all the edge data objects in the graph

**Return type** list

### `extend_from_edge_list(edge_list, /)`

Extend graph from an edge list



This method differs from `add_edges_from_no_data()` in that it will add nodes if a node index is not present in the edge list.

**Parameters** `edge_list` (*list*) – A list of tuples of the form (source, target) where source and target are integer node indices. If the node index is not present in the graph, nodes will be added (with a node weight of None) to that index.

**extend\_from\_weighted\_edge\_list**(*edge\_list*, /)

Extend graph from a weighted edge list

This method differs from `add_edges_from()` in that it will add nodes if a node index is not present in the edge list.

**Parameters** `edge_list` (*list*) – A list of tuples of the form (source, target, weight) where source and target are integer node indices. If the node index is not present in the graph nodes will be added (with a node weight of None) to that index.

**find\_adjacent\_node\_by\_edge**(*node*, *predicate*, /)

Find a target node with a specific edge

This method is used to find a target node that is adjacent to a given node given an edge condition.

**Parameters**

- **node** (*int*) – The node to use as the source of the search
- **predicate** (*callable*) – A python callable that will take a single parameter, the edge object, and will return a boolean if the edge matches or not

**Returns** The node object that has an edge to it from the provided node index which matches the provided condition

**find\_node\_by\_weight**()

Find node within this graph given a specific weight

This algorithm has a worst case of O(n) since it searches the node indices in order. If there is more than one node in the graph with the same weight only the first match (by node index) will be returned.

**Parameters** `obj` – The weight to look for in the graph.

**Returns** the index of the first node in the graph that is equal to the weight. If no match is found None will be returned.

**Return type** int

**static from\_adjacency\_matrix**(*matrix*, /)

Create a new `PyDiGraph` object from an adjacency matrix

This method can be used to construct a new `PyDiGraph` object from an input adjacency matrix. The node weights will be the index from the matrix. The edge weights will be a float value of the value from the matrix.

**Parameters** `matrix` (*ndarray*) – The input numpy array adjacency matrix to create a new `PyDiGraph` object from. It must be a 2 dimensional array and be a float/np.float64 data type.

**Returns** A new graph object generated from the adjacency matrix

**Return type** `PyDiGraph`

**get\_all\_edge\_data**(*node\_a*, *node\_b*, /)

Return the edge data for all the edges between 2 nodes.

**Parameters**

- **node\_a** (*int*) – The index for the first node
- **node\_b** (*int*) – The index for the second node

**Returns** A list with all the data objects for the edges between nodes

**Return type** list

**Raises** *NoEdgeBetweenNodes* – When there is no edge between nodes

**get\_edge\_data**(*node\_a, node\_b, /*)

Return the edge data for an edge between 2 nodes.

**Parameters**

- **node\_a** (*int*) – The index for the first node
- **node\_b** (*int*) – The index for the second node

**Returns** The data object set for the edge

**Raises** *NoEdgeBetweenNodes* – When there is no edge between nodes

**get\_node\_data**(*node, /*)

Return the node data for a given node index

**Parameters** **node** (*int*) – The index for the node

**Returns** The data object set for that node

**Raises** *IndexError* – when an invalid node index is provided

**has\_edge**(*node\_a, node\_b, /*)

Return True if there is an edge from node\_a to node\_b.

**Parameters**

- **node\_a** (*int*) – The source node index to check for an edge
- **node\_b** (*int*) – The destination node index to check for an edge

**Returns** True if there is an edge false if there is no edge

**Return type** bool

**in\_degree**(*node, /*)

Get the degree of a node for inbound edges.

**Parameters** **node** (*int*) – The index of the node to find the inbound degree of

**Returns** The inbound degree for the specified node

**Return type** int

**in\_edges**(*node, /*)

Get the index and edge data for all parents of a node.

This will return a list of tuples with the parent index the node index and the edge data. This can be used to recreate `add_edge()` calls. :param int node: The index of the node to get the edges for

**Parameters** **node** (*int*) – The index of the node to get the edges for

**Returns** A list of tuples of the form: (parent\_index, node\_index, edge\_data)`

**Return type** *WeightedEdgeList*

**insert\_node\_on\_in\_edges**(*node, ref\_node, /*)

Insert a node between a reference node and all its predecessor nodes

This essentially iterates over all edges into the reference node specified in the `ref_node` parameter removes those edges and then adds 2 edges, one from the predecessor of `ref_node` to `node` and the other from `node` to `ref_node`. The edge payloads for the newly created edges are copied by reference from the original edge that gets removed.

**Parameters**

- **node** (*int*) – The node index to insert between
- **ref\_node** (*int*) – The reference node index to insert node between

**insert\_node\_on\_in\_edges\_multiple**(*node, ref\_nodes, /*)

Insert a node between a list of reference nodes and all their predecessors

This essentially iterates over all edges into the reference node specified in the `ref_nodes` parameter removes those edges and then adds 2 edges, one from the predecessor of `ref_node` to `node` and the other from `node` to `ref_node`. The edge payloads for the newly created edges are copied by reference from the original edge that gets removed.

**Parameters**

- **node** (*int*) – The node index to insert between
- **ref\_node** (*int*) – The reference node index to insert node between

**insert\_node\_on\_out\_edges**(*node, ref\_node, /*)

Insert a node between a reference node and all its successor nodes

This essentially iterates over all edges out of the reference node specified in the `ref_node` parameter removes those edges and then adds 2 edges, one from `ref_node` to `node` and the other from `node` to the successor of `ref_node`. The edge payloads for the newly created edges are copied by reference from the original edge that gets removed.

**Parameters**

- **node** (*int*) – The node index to insert between
- **ref\_node** (*int*) – The reference node index to insert node between

**insert\_node\_on\_out\_edges\_multiple**(*node, ref\_nodes, /*)

Insert a node between a list of reference nodes and all their successors

This essentially iterates over all edges out of the reference node specified in the `ref_node` parameter removes those edges and then adds 2 edges, one from `ref_node` to `node` and the other from `node` to the successor of `ref_node`. The edge payloads for the newly created edges are copied by reference from the original edge that gets removed.

**Parameters**

- **node** (*int*) – The node index to insert between
- **ref\_nodes** (*int*) – The list of node indices to insert node between

**is\_symmetric**()

Check if the graph is symmetric

**Returns** True if the graph is symmetric

**Return type** bool

**merge\_nodes**(*u, /, v*)

Merge two nodes in the graph.

If the nodes have equal weight objects then all the edges into and out of  $u$  will be added to  $v$  and  $u$  will be removed from the graph. If the nodes don't have equal weight objects then no changes will be made and no error raised

**Parameters**

- **u** (*int*) – The source node that is going to be merged
- **v** (*int*) – The target node that is going to be the new node

**multigraph**

Whether the graph is a multigraph (allows multiple edges between nodes) or not

If set to `False` multiple edges between nodes are not allowed and calls that would add a parallel edge will instead update the existing edge

**neighbors**(*node*, /)

Get the neighbors (i.e. successors) of a node.

This will return a list of neighbor node indices. This function is equivalent to `successor_indices()`.

**Parameters** **node** (*int*) – The index of the node to get the neighbors of

**Returns** A list of the neighbor node indices

**Return type** *NodeIndices*

**node\_indexes**()

Return a list of all node indexes.

**Returns** A list of all the node indexes in the graph

**Return type** *NodeIndices*

**nodes**()

Return a list of all node data.

**Returns** A list of all the node data objects in the graph

**Return type** list

**out\_degree**(*node*, /)

Get the degree of a node for outbound edges.

**Parameters** **node** (*int*) – The index of the node to find the outbound degree of

**Returns** The outbound degree for the specified node

**Return type** int

**out\_edges**(*node*, /)

Get the index and edge data for all children of a node.

This will return a list of tuples with the child index the node index and the edge data. This can be used to recreate `add_edge()` calls.

**Parameters** **node** (*int*) – The index of the node to get the edges for

**Returns** **out\_edges** A list of tuples of the form: ``(node_index, child_index, edge_data)``

**Return type** *WeightedEdgeList*

**predecessor\_indices**(*node*, /)

Get the predecessor indices of a node.

This will return a list of the node indices for the predecessors of a node

**Parameters** `node` (*int*) – The index of the node to get the predecessors of

**Returns** A list of the neighbor node indices

**Return type** *NodeIndices*

**predecessors**(*node, /*)

Return a list of all the node predecessor data.

**Parameters** `node` (*int*) – The index for the node to get the predecessors for

**Returns** A list of the node data for all the parent neighbor nodes

**Return type** list

**static read\_edge\_list**(*path, /, comment=None, delimitator=None*)

Read an edge list file and create a new PyDiGraph object from the contents

The expected format for the edge list file is a line separated list of delimited node ids. If there are more than 3 elements on a line the 3rd on will be treated as a string weight for the edge

**Parameters**

- **path** (*str*) – The path of the file to open
- **comment** (*str*) – Optional character to use as a comment by default there are no comment characters
- **delimitator** (*str*) – Optional character to use as a delimitator by default any whitespace will be used

For example:

```
import os
import tempfile

from PIL import Image
import pydot

import networkx

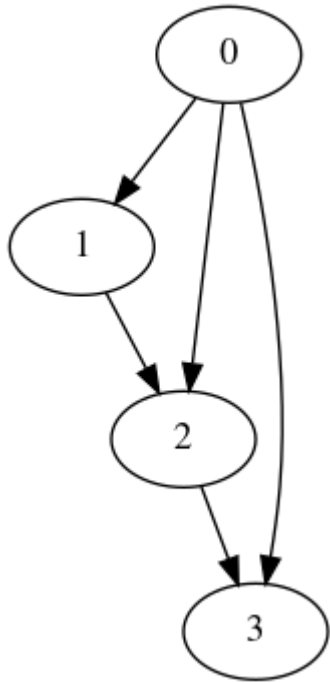
with tempfile.NamedTemporaryFile('wt') as fd:
    path = fd.name
    fd.write('0 1\n')
    fd.write('0 2\n')
    fd.write('0 3\n')
    fd.write('1 2\n')
    fd.write('2 3\n')
    fd.flush()
    graph = networkx.PyDiGraph.read_edge_list(path)

# Draw graph
dot = pydot.graph_from_dot_data(graph.to_dot())[0]

with tempfile.TemporaryDirectory() as tmpdirname:
    tmp_path = os.path.join(tmpdirname, 'dag.png')
    dot.write_png(tmp_path)
    image = Image.open(tmp_path)
```

(continues on next page)

```
os.remove(tmp_path)
image
```



**remove\_edge**(*parent*, *child*, /)

Remove an edge between 2 nodes.

Note if there are multiple edges between the specified nodes only one will be removed.

**Parameters**

- **parent** (*int*) – The index for the parent node.
- **child** (*int*) – The index of the child node.

**Raises** *NoEdgeBetweenNodes* – If there are no edges between the nodes specified

**remove\_edge\_from\_index**(*edge*, /)

Remove an edge identified by the provided index

**Parameters** **edge** (*int*) – The index of the edge to remove

**remove\_edges\_from**(*index\_list*, /)

Remove edges from the graph.

Note if there are multiple edges between the specified nodes only one will be removed.

**Parameters** **index\_list** (*list*) – A list of node index pairs to remove from the graph

**remove\_node**(*node*, /)

Remove a node from the graph.

**Parameters** **node** (*int*) – The index of the node to remove. If the index is not present in the graph it will be ignored and this function will have no effect.

**remove\_node\_retain\_edges**(*node*, /, *use\_outgoing=None*, *condition=None*)

Remove a node from the graph and add edges from all predecessors to all successors

By default the data/weight on edges into the removed node will be used for the retained edges.

#### Parameters

- **node** (*int*) – The index of the node to remove. If the index is not present in the graph it will be ignored and this function will have no effect.
- **use\_outgoing** (*bool*) – If set to true the weight/data from the edge outgoing from node will be used in the retained edge instead of the default weight/data from the incoming edge.
- **condition** – A callable that will be passed 2 edge weight/data objects, one from the incoming edge to **node** the other for the outgoing edge, and will return a *bool* on whether an edge should be retained. For example setting this kwarg to:

```
lambda in_edge, out_edge: in_edge == out_edge
```

would only retain edges if the input edge to node had the same data payload as the outgoing edge.

**remove\_nodes\_from**(*index\_list, /*)

Remove nodes from the graph.

If a node index in the list is not present in the graph it will be ignored.

**Parameters** **index\_list** (*list*) – A list of node indices to remove from the graph.

**subgraph**(*nodes, /*)

Return a new PyDiGraph object for a subgraph of this graph

**Parameters** **nodes** (*list*) – A list of node indices to generate the subgraph from. If a node index is included that is not present in the graph it will silently be ignored.

**Returns** A new PyDiGraph object representing a subgraph of this graph. It is worth noting that node and edge weight/data payloads are passed by reference so if you update (not replace) an object used as the weight in graph or the subgraph it will also be updated in the other.

**Return type** *PyGraph*

**successor\_indices**(*node, /*)

Get the successor indices of a node.

This will return a list of the node indices for the successors of a node

**Parameters** **node** (*int*) – The index of the node to get the successors of

**Returns** A list of the neighbor node indices

**Return type** *NodeIndices*

**successors**(*node, /*)

Return a list of all the node successor data.

**Parameters** **node** (*int*) – The index for the node to get the successors for

**Returns** A list of the node data for all the child neighbor nodes

**Return type** *list*

**to\_dot**(*node\_attr=None, edge\_attr=None, graph\_attr=None, filename=None*)

Generate a dot file from the graph

#### Parameters

- **node\_attr** – A callable that will take in a node data object and return a dictionary of attributes to be associated with the node in the dot file. The key and value of this dictionary

**must** be strings. If they're not strings networkx will raise `TypeError` (unfortunately without an error message because of current limitations in the PyO3 type checking)

- **edge\_attr** – A callable that will take in an edge data object and return a dictionary of attributes to be associated with the node in the dot file. The key and value of this dictionary **must** be a string. If they're not strings networkx will raise `TypeError` (unfortunately without an error message because of current limitations in the PyO3 type checking)
- **graph\_attr** (*dict*) – An optional dictionary that specifies any graph attributes for the output dot file. The key and value of this dictionary **must** be a string. If they're not strings networkx will raise `TypeError` (unfortunately without an error message because of current limitations in the PyO3 type checking)
- **filename** (*str*) – An optional path to write the dot file to if specified there is no return from the function

**Returns** A string with the dot file contents if filename is not specified.

**Return type** `str`

Using this method enables you to leverage graphviz to visualize a `networkx.PyDiGraph` object. For example:

```
import os
import tempfile

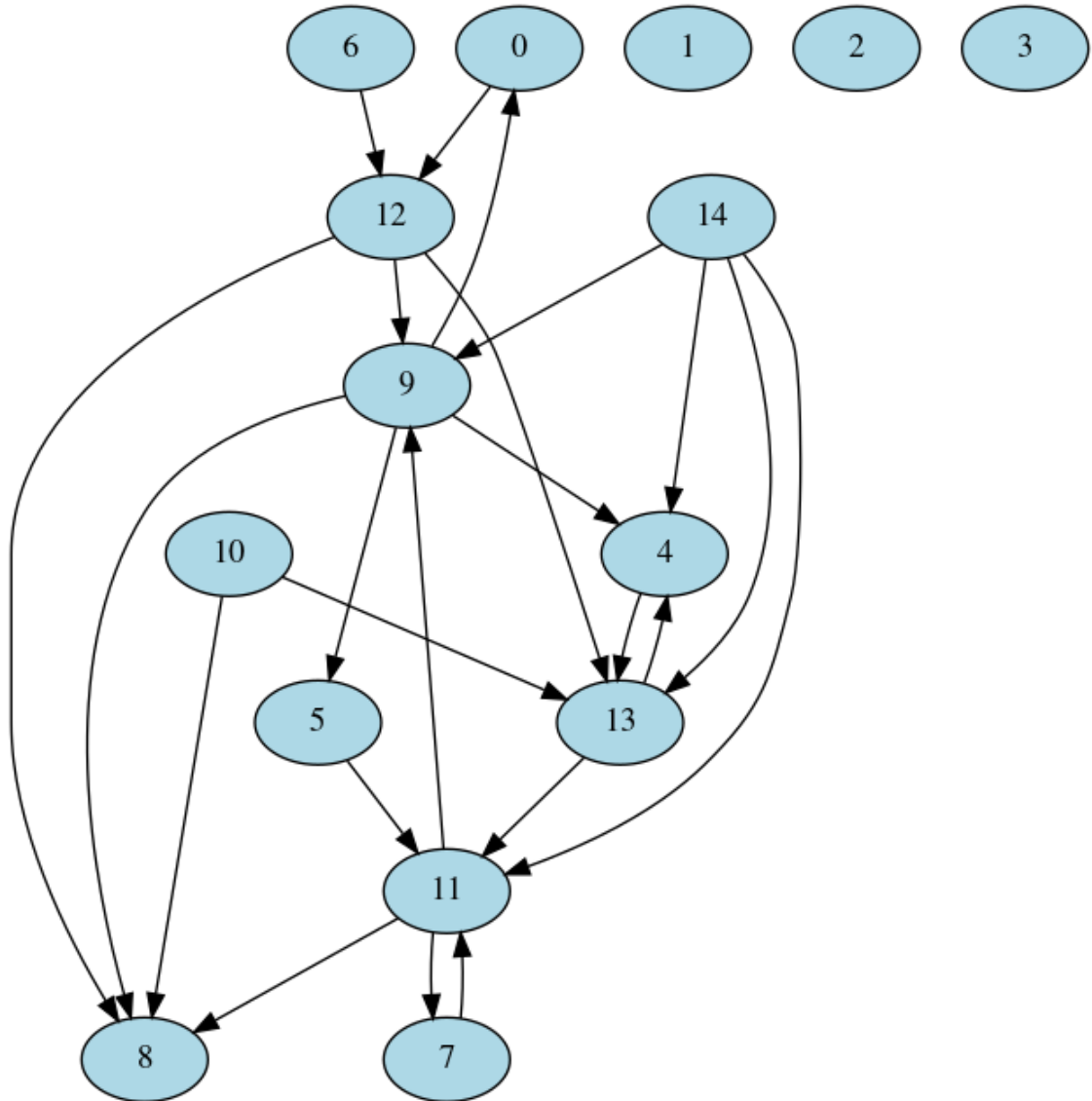
import pydot
from PIL import Image

import networkx

graph = networkx.directed_gnp_random_graph(15, .25)
dot_str = graph.to_dot(
    lambda node: dict(
        color='black', fillcolor='lightblue', style='filled'))
dot = pydot.graph_from_dot_data(dot_str)[0]

with tempfile.TemporaryDirectory() as tmpdirname:
    tmp_path = os.path.join(tmpdirname, 'dag.png')
    dot.write_png(tmp_path)
    image = Image.open(tmp_path)
    os.remove(tmp_path)
image
```



**to\_undirected()**

Generate a new PyGraph object from this graph

This will create a new *PyGraph* object from this graph. All edges in this graph will be created as undirected edges in the new graph object. Do note that the node and edge weights/data payloads will be passed by reference to the new *PyGraph* object.

**Returns** A new PyGraph object with an undirected edge for every directed edge in this graph

**Return type** *PyGraph*

**update\_edge(source, target, /, edge)**

Update an edge's weight/payload inplace

If there are parallel edges in the graph only one edge will be updated. if you need to update a specific edge or need to ensure all parallel edges get updated you should use *update\_edge\_by\_index()* instead.

**Parameters**

- **source** (*int*) – The index for the first node

- **target** (*int*) – The index for the second node

**Raises** *NoEdgeBetweenNodes* – When there is no edge between nodes

**update\_edge\_by\_index**(*source, target, /, edge*)

Update an edge’s weight/payload by the edge index

**Parameters**

- **edge\_index** (*int*) – The index for the edge
- **edge** (*object*) – The data payload/weight to update the edge with

**Raises** *NoEdgeBetweenNodes* – When there is no edge between nodes

**weighted\_edge\_list**()

Get edge list with weights

Returns a list of tuples of the form (*source, target, weight*) where *source* and *target* are the node indices and *weight* is the payload of the edge.

**Returns** An edge list with weights

**Return type** *WeightedEdgeList*

## 2.2 Generators

<code>networkx.generators.cycle_graph</code> ( <i>num_nodes, ...</i> )	Generate an undirected cycle graph
<code>networkx.generators.directed_cycle_graph</code> ( <i>...</i> )	Generate a cycle graph
<code>networkx.generators.path_graph</code> ( <i>num_nodes, ...</i> )	Generate an undirected path graph
<code>networkx.generators.directed_path_graph</code> ( <i>...</i> )	Generate a directed path graph
<code>networkx.generators.star_graph</code> ( <i>num_nodes, ...</i> )	Generate an undirected star graph
<code>networkx.generators.directed_star_graph</code> ( <i>...</i> )	Generate a directed star graph
<code>networkx.generators.mesh_graph</code> ( <i>num_nodes, ...</i> )	Generate an undirected mesh graph where every node is connected to every other
<code>networkx.generators.directed_mesh_graph</code> ( <i>...</i> )	Generate a directed mesh graph where every node is connected to every other
<code>networkx.generators.grid_graph</code> ( <i>rows, cols, ...</i> )	Generate an undirected grid graph.
<code>networkx.generators.directed_grid_graph</code> ( <i>...</i> )	Generate a directed grid graph. The edges propagate towards right and

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## 2.2.1 networkx.generators.cycle\_graph

**cycle\_graph**(*num\_nodes=None, weights=None, multigraph=True, /*)

Generate an undirected cycle graph

### Parameters

- **num\_node** (*int*) – The number of nodes to generate the graph with. Node weights will be None if this is specified. If both `num_node` and `weights` are set this will be ignored and `weights` will be used.
- **weights** (*list*) – A list of node weights, the first element in the list will be the center node of the cycle graph. If both `num_node` and `weights` are set this will be ignored and `weights` will be used.
- **multigraph** (*bool*) – When set to False the output *PyGraph* object will not be a multigraph and won't allow parallel edges to be added. Instead calls which would create a parallel edge will update the existing edge.

**Returns** The generated cycle graph

**Return type** *PyGraph*

**Raises** **IndexError** – If neither `num_nodes` or `weights` are specified

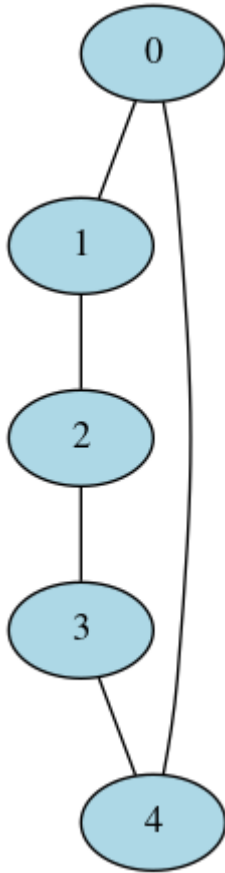
```
import os
import tempfile

import pydot
from PIL import Image

import networkx.generators

graph = networkx.generators.cycle_graph(5)
dot_str = graph.to_dot(
    lambda node: dict(
        color='black', fillcolor='lightblue', style='filled'))
dot = pydot.graph_from_dot_data(dot_str)[0]

with tempfile.TemporaryDirectory() as tmpdirname:
    tmp_path = os.path.join(tmpdirname, 'dag.png')
    dot.write_png(tmp_path)
    image = Image.open(tmp_path)
    os.remove(tmp_path)
image
```



**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

### 2.2.2 networkx.generators.directed\_cycle\_graph

**directed\_cycle\_graph**(*num\_nodes=None, weights=None, bidirectional=False, /*)

Generate a cycle graph

#### Parameters

- **num\_node** (*int*) – The number of nodes to generate the graph with. Node weights will be None if this is specified. If both `num_node` and `weights` are set this will be ignored and `weights` will be used.
- **weights** (*list*) – A list of node weights, the first element in the list will be the center node of the cycle graph. If both `num_node` and `weights` are set this will be ignored and `weights` will be used.
- **bidirectional** (*bool*) – Adds edges in both directions between two nodes if set to True. Default value is False

**Returns** The generated cycle graph

**Return type** *PyDiGraph*

Raises `IndexError` – If neither `num_nodes` or `weights` are specified

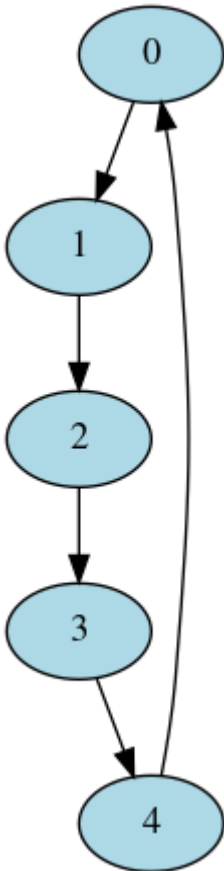
```
import os
import tempfile

import pydot
from PIL import Image

import networkx.generators

graph = networkx.generators.directed_cycle_graph(5)
dot_str = graph.to_dot(
    lambda node: dict(
        color='black', fillcolor='lightblue', style='filled'))
dot = pydot.graph_from_dot_data(dot_str)[0]

with tempfile.TemporaryDirectory() as tmpdirname:
    tmp_path = os.path.join(tmpdirname, 'dag.png')
    dot.write_png(tmp_path)
    image = Image.open(tmp_path)
    os.remove(tmp_path)
image
```



**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

### 2.2.3 networkx.generators.path\_graph

**path\_graph**(*num\_nodes=None, weights=None, multigraph=True, /*)

Generate an undirected path graph

#### Parameters

- **num\_node** (*int*) – The number of nodes to generate the graph with. Node weights will be None if this is specified. If both `num_node` and `weights` are set this will be ignored and `weights` will be used.
- **weights** (*list*) – A list of node weights, the first element in the list will be the center node of the path graph. If both `num_node` and `weights` are set this will be ignored and `weights` will be used.
- **multigraph** (*bool*) – When set to False the output *PyGraph* object will not be not be a multigraph and won't allow parallel edges to be added. Instead calls which would create a parallel edge will update the existing edge.

**Returns** The generated path graph

**Return type** *PyGraph*

**Raises** **IndexError** – If neither `num_nodes` or `weights` are specified

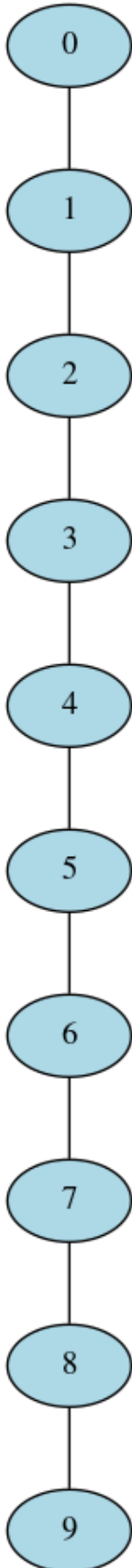
```
import os
import tempfile

import pydot
from PIL import Image

import networkx.generators

graph = networkx.generators.path_graph(10)
dot_str = graph.to_dot(
    lambda node: dict(
        color='black', fillcolor='lightblue', style='filled'))
dot = pydot.graph_from_dot_data(dot_str)[0]

with tempfile.TemporaryDirectory() as tmpdirname:
    tmp_path = os.path.join(tmpdirname, 'dag.png')
    dot.write_png(tmp_path)
    image = Image.open(tmp_path)
    os.remove(tmp_path)
image
```



**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## 2.2.4 networkx.generators.directed\_path\_graph

**directed\_path\_graph**(*num\_nodes=None, weights=None, bidirectional=False, /*)

Generate a directed path graph

### Parameters

- **num\_node** (*int*) – The number of nodes to generate the graph with. Node weights will be None if this is specified. If both `num_node` and `weights` are set this will be ignored and `weights` will be used.
- **weights** (*list*) – A list of node weights, the first element in the list will be the center node of the path graph. If both `num_node` and `weights` are set this will be ignored and `weights` will be used.
- **bidirectional** (*bool*) – Adds edges in both directions between two nodes if set to True. Default value is False

**Returns** The generated path graph

**Return type** *PyDiGraph*

**Raises** **IndexError** – If neither `num_nodes` or `weights` are specified

```
import os
import tempfile

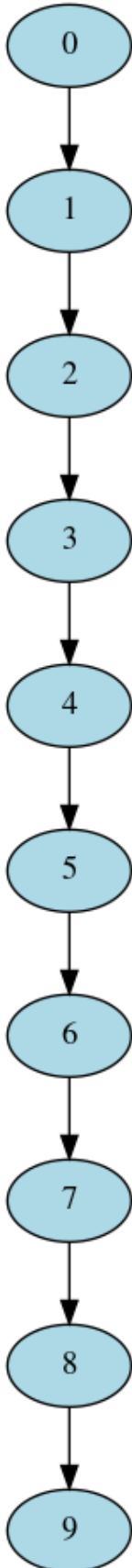
import pydot
from PIL import Image

import networkx.generators

graph = networkx.generators.directed_path_graph(10)
dot_str = graph.to_dot(
    lambda node: dict(
        color='black', fillcolor='lightblue', style='filled'))
dot = pydot.graph_from_dot_data(dot_str)[0]

with tempfile.TemporaryDirectory() as tmpdirname:
    tmp_path = os.path.join(tmpdirname, 'dag.png')
    dot.write_png(tmp_path)
    image = Image.open(tmp_path)
    os.remove(tmp_path)
image
```





**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## 2.2.5 networkx.generators.star\_graph

**star\_graph**(*num\_nodes=None, weights=None, multigraph=True, /*)

Generate an undirected star graph

### Parameters

- **num\_node** (*int*) – The number of nodes to generate the graph with. Node weights will be None if this is specified. If both `num_node` and `weights` are set this will be ignored and `weights` will be used.
- **weights** (*list*) – A list of node weights, the first element in the list will be the center node of the star graph. If both `num_node` and `weights` are set this will be ignored and `weights` will be used.
- **multigraph** (*bool*) – When set to False the output *PyGraph* object will not be not be a multigraph and won't allow parallel edges to be added. Instead calls which would create a parallel edge will update the existing edge.

**Returns** The generated star graph

**Return type** *PyGraph*

**Raises** **IndexError** – If neither `num_nodes` or `weights` are specified

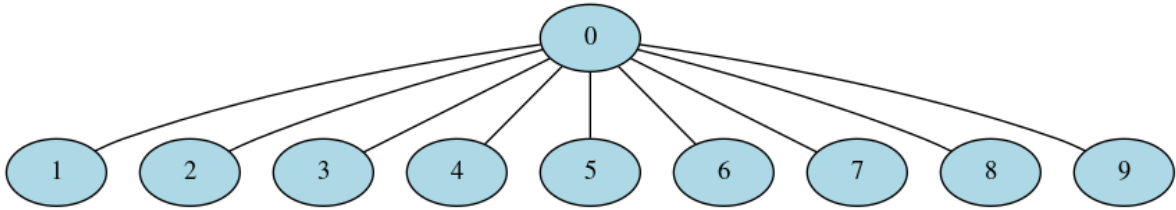
```
import os
import tempfile

import pydot
from PIL import Image

import networkx.generators

graph = networkx.generators.star_graph(10)
dot_str = graph.to_dot(
    lambda node: dict(
        color='black', fillcolor='lightblue', style='filled'))
dot = pydot.graph_from_dot_data(dot_str)[0]

with tempfile.TemporaryDirectory() as tmpdirname:
    tmp_path = os.path.join(tmpdirname, 'dag.png')
    dot.write_png(tmp_path)
    image = Image.open(tmp_path)
    os.remove(tmp_path)
image
```



**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## 2.2.6 networkx.generators.directed\_star\_graph

**directed\_star\_graph**(*num\_nodes=None, weights=None, inward=False, bidirectional=False, /*)

Generate a directed star graph

### Parameters

- **num\_node** (*int*) – The number of nodes to generate the graph with. Node weights will be None if this is specified. If both `num_node` and `weights` are set this will be ignored and `weights` will be used.
- **weights** (*list*) – A list of node weights, the first element in the list will be the center node of the star graph. If both `num_node` and `weights` are set this will be ignored and `weights` will be used.
- **bidirectional** (*bool*) – Adds edges in both directions between two nodes if set to True. Default value is False.
- **inward** (*bool*) – If set True the nodes will be directed towards the center node. This parameter is ignored if `bidirectional` is set to True.

**Returns** The generated star graph

**Return type** *PyDiGraph*

**Raises** **IndexError** – If neither `num_nodes` or `weights` are specified

```

import os
import tempfile

import pydot
from PIL import Image

import networkx.generators

graph = networkx.generators.directed_star_graph(10)
dot_str = graph.to_dot(
    lambda node: dict(
        color='black', fillcolor='lightblue', style='filled'))
dot = pydot.graph_from_dot_data(dot_str)[0]

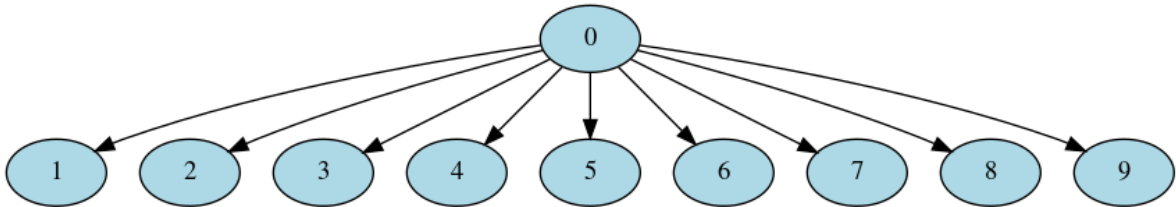
with tempfile.TemporaryDirectory() as tmpdirname:

```

(continues on next page)

(continued from previous page)

```
tmp_path = os.path.join(tmpdirname, 'dag.png')
dot.write_png(tmp_path)
image = Image.open(tmp_path)
os.remove(tmp_path)
image
```



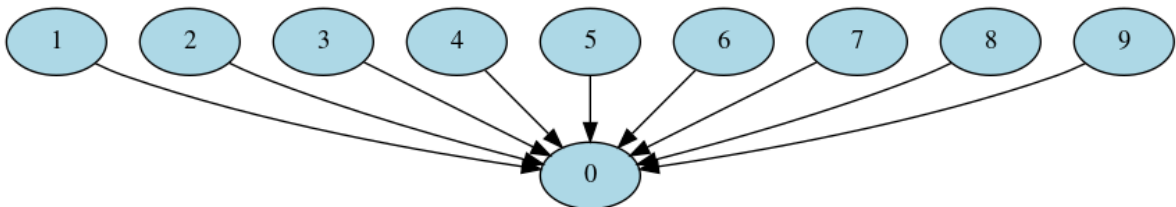
```
import os
import tempfile

import pydot
from PIL import Image

import networkx.generators

graph = networkx.generators.directed_star_graph(10, inward=True)
dot_str = graph.to_dot(
    lambda node: dict(
        color='black', fillcolor='lightblue', style='filled'))
dot = pydot.graph_from_dot_data(dot_str)[0]

with tempfile.TemporaryDirectory() as tmpdirname:
    tmp_path = os.path.join(tmpdirname, 'dag.png')
    dot.write_png(tmp_path)
    image = Image.open(tmp_path)
    os.remove(tmp_path)
image
```



**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## 2.2.7 networkx.generators.mesh\_graph

**mesh\_graph**(*num\_nodes=None, weights=None, multigraph=True, /*)

Generate an undirected mesh graph where every node is connected to every other

### Parameters

- **num\_node** (*int*) – The number of nodes to generate the graph with. Node weights will be None if this is specified. If both `num_node` and `weights` are set this will be ignored and `weights` will be used.
- **weights** (*list*) – A list of node weights. If both `num_node` and `weights` are set this will be ignored and `weights` will be used.
- **multigraph** (*bool*) – When set to False the output *PyGraph* object will not be not be a multigraph and won't allow parallel edges to be added. Instead calls which would create a parallel edge will update the existing edge.

**Returns** The generated mesh graph

**Return type** *PyGraph*

**Raises** **IndexError** – If neither `num_nodes` or `weights` are specified

```
import os
import tempfile

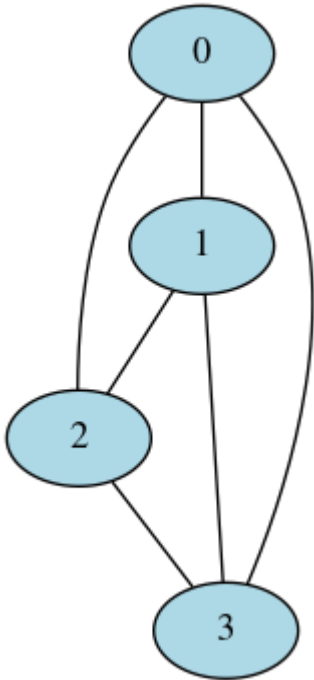
import pydot
from PIL import Image

import networkx.generators

graph = networkx.generators.mesh_graph(4)
dot_str = graph.to_dot(
    lambda node: dict(
        color='black', fillcolor='lightblue', style='filled'))
dot = pydot.graph_from_dot_data(dot_str)[0]

with tempfile.TemporaryDirectory() as tmpdirname:
    tmp_path = os.path.join(tmpdirname, 'dag.png')
    dot.write_png(tmp_path)
    image = Image.open(tmp_path)
    os.remove(tmp_path)

image
```



**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## 2.2.8 networkx.generators.directed\_mesh\_graph

`directed_mesh_graph(num_nodes=None, weights=None, /)`

Generate a directed mesh graph where every node is connected to every other

### Parameters

- **num\_node** (*int*) – The number of nodes to generate the graph with. Node weights will be None if this is specified. If both `num_node` and `weights` are set this will be ignored and `weights` will be used.
- **weights** (*list*) – A list of node weights. If both `num_node` and `weights` are set this will be ignored and `weights` will be used.

**Returns** The generated mesh graph

**Return type** *PyDiGraph*

**Raises** **IndexError** – If neither `num_nodes` or `weights` are specified

```
import os
import tempfile

import pydot
from PIL import Image
```

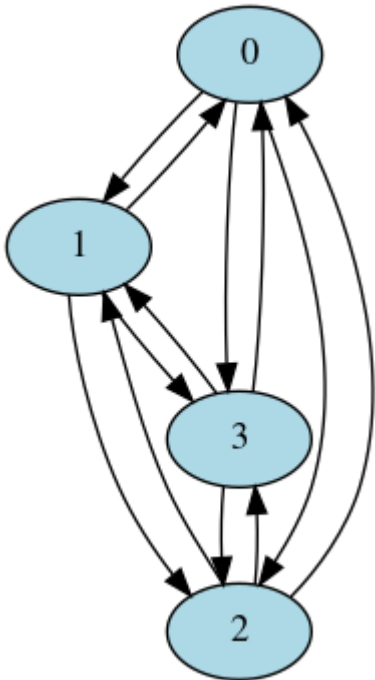
(continues on next page)

(continued from previous page)

```
import networkx.generators

graph = networkx.generators.directed_mesh_graph(4)
dot_str = graph.to_dot(
    lambda node: dict(
        color='black', fillcolor='lightblue', style='filled'))
dot = pydot.graph_from_dot_data(dot_str)[0]

with tempfile.TemporaryDirectory() as tmpdirname:
    tmp_path = os.path.join(tmpdirname, 'dag.png')
    dot.write_png(tmp_path)
    image = Image.open(tmp_path)
    os.remove(tmp_path)
image
```



**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## 2.2.9 networkx.generators.grid\_graph

**grid\_graph**(rows=None, cols=None, weights=None, multigraph=True, /)

Generate an undirected grid graph.

### Parameters

- **rows** (*int*) – The number of rows to generate the graph with. If specified, cols also need to be specified
- **cols** (*list*) – The number of rows to generate the graph with. If specified, rows also need to be specified. rows\*cols defines the number of nodes in the graph
- **weights** (*list*) – A list of node weights. Nodes are filled row wise. If rows and cols are not specified, then a linear graph containing all the values in weights list is created. If number of nodes(rows\*cols) is less than length of weights list, the trailing weights are ignored. If number of nodes(rows\*cols) is greater than length of weights list, extra nodes with None weight are appended.
- **multigraph** (*bool*) – When set to False the output *PyGraph* object will not be a multigraph and won't allow parallel edges to be added. Instead calls which would create a parallel edge will update the existing edge.

**Returns** The generated grid graph

**Return type** *PyGraph*

**Raises** **IndexError** – If neither rows or cols and weights are specified

```
import os
import tempfile

import pydot
from PIL import Image

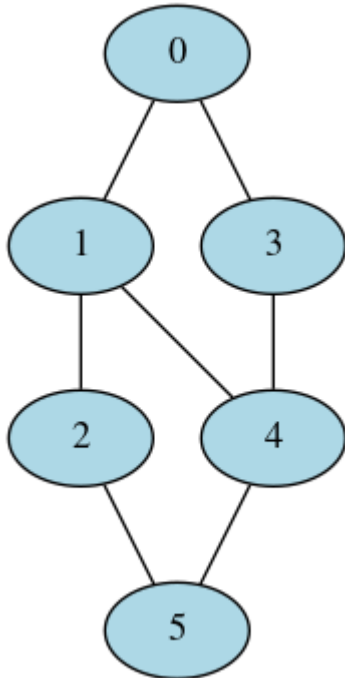
import networkx.generators

graph = networkx.generators.grid_graph(2, 3)
dot_str = graph.to_dot(
    lambda node: dict(
        color='black', fillcolor='lightblue', style='filled'))
dot = pydot.graph_from_dot_data(dot_str)[0]

with tempfile.TemporaryDirectory() as tmpdirname:
    tmp_path = os.path.join(tmpdirname, 'dag.png')
    dot.write_png(tmp_path)
    image = Image.open(tmp_path)
    os.remove(tmp_path)

image
```





**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

### 2.2.10 networkx.generators.directed\_grid\_graph

`directed_grid_graph(rows=None, cols=None, weights=None, bidirectional=False, /)`

Generate a directed grid graph. The edges propagate towards right and bottom direction if `bidirectional` is false

#### Parameters

- **rows** (*int*) – The number of rows to generate the graph with. If specified, cols also need to be specified.
- **cols** (*list*) – The number of rows to generate the graph with. If specified, rows also need to be specified. `rows*cols` defines the number of nodes in the graph.
- **weights** (*list*) – A list of node weights. Nodes are filled row wise. If rows and cols are not specified, then a linear graph containing all the values in weights list is created. If number of nodes(`rows*cols`) is less than length of weights list, the trailing weights are ignored. If number of nodes(`rows*cols`) is greater than length of weights list, extra nodes with None weight are appended.
- **bidirectional** – A parameter to indicate if edges should exist in both directions between nodes

**Returns** The generated grid graph

Return type *PyDiGraph*

Raises **IndexError** – If neither rows or cols and weights are specified

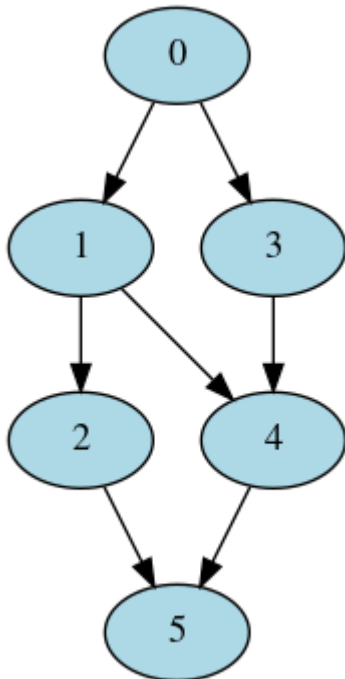
```
import os
import tempfile

import pydot
from PIL import Image

import networkx.generators

graph = networkx.generators.directed_grid_graph(2, 3)
dot_str = graph.to_dot(
    lambda node: dict(
        color='black', fillcolor='lightblue', style='filled'))
dot = pydot.graph_from_dot_data(dot_str)[0]

with tempfile.TemporaryDirectory() as tmpdirname:
    tmp_path = os.path.join(tmpdirname, 'dag.png')
    dot.write_png(tmp_path)
    image = Image.open(tmp_path)
    os.remove(tmp_path)
image
```



## 2.3 Random Circuit Functions

<code>networkx.directed_gnp_random_graph(...[, seed])</code>	Return a $G_{np}$ directed random graph, also known as an Erdős-Rényi graph or a binomial graph.
<code>networkx.undirected_gnp_random_graph(...[, seed])</code>	Return a $G_{np}$ random undirected graph, also known as an Erdős-Rényi graph or a binomial graph.
<code>networkx.directed_gnm_random_graph(...[, seed])</code>	Return a $G_{nm}$ of a directed graph
<code>networkx.undirected_gnm_random_graph(...[, seed])</code>	Return a $G_{nm}$ of an undirected graph

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

### 2.3.1 networkx.directed\_gnp\_random\_graph

**directed\_gnp\_random\_graph**(*num\_nodes*, *probability*, *seed=None*, /)

Return a  $G_{np}$  directed random graph, also known as an Erdős-Rényi graph or a binomial graph.

For number of nodes  $n$  and probability  $p$ , the  $G_{n,p}$  graph algorithm creates  $n$  nodes, and for all the  $n(n-1)$  possible edges, each edge is created independently with probability  $p$ . In general, for any probability  $p$ , the expected number of edges returned is  $m = pn(n-1)$ . If  $p = 0$  or  $p = 1$ , the returned graph is not random and will always be an empty or a complete graph respectively. An empty graph has zero edges and a complete directed graph has  $n(n-1)$  edges. The run time is  $O(n+m)$  where  $m$  is the expected number of edges mentioned above. When  $p = 0$ , run time always reduces to  $O(n)$ , as the lower bound. When  $p = 1$ , run time always goes to  $O(n + n(n-1))$ , as the upper bound. For other probabilities, this algorithm<sup>1</sup> runs in  $O(n+m)$  time.

For  $0 < p < 1$ , the algorithm is based on the implementation of the networkx function `fast_gnp_random_graph`<sup>2</sup>

#### Parameters

- **num\_nodes** (*int*) – The number of nodes to create in the graph
- **probability** (*float*) – The probability of creating an edge between two nodes
- **seed** (*int*) – An optional seed to use for the random number generator

**Returns** A PyDiGraph object

**Return type** *PyDiGraph*

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

<sup>1</sup> Vladimir Batagelj and Ulrik Brandes, “Efficient generation of large random networks”, Phys. Rev. E, 71, 036113, 2005.

<sup>2</sup> [https://github.com/networkx/networkx/blob/networkx-2.4/networkx/generators/random\\_graphs.py#L49-L120](https://github.com/networkx/networkx/blob/networkx-2.4/networkx/generators/random_graphs.py#L49-L120)

## 2.3.2 networkx.undirected\_gnp\_random\_graph

**undirected\_gnp\_random\_graph**(*num\_nodes*, *probability*, *seed=None, /*)

Return a  $G_{np}$  random undirected graph, also known as an Erdős-Rényi graph or a binomial graph.

For number of nodes  $n$  and probability  $p$ , the  $G_{n,p}$  graph algorithm creates  $n$  nodes, and for all the  $n(n-1)/2$  possible edges, each edge is created independently with probability  $p$ . In general, for any probability  $p$ , the expected number of edges returned is  $m = pn(n-1)/2$ . If  $p = 0$  or  $p = 1$ , the returned graph is not random and will always be an empty or a complete graph respectively. An empty graph has zero edges and a complete undirected graph has  $n(n-1)/2$  edges. The run time is  $O(n+m)$  where  $m$  is the expected number of edges mentioned above. When  $p = 0$ , run time always reduces to  $O(n)$ , as the lower bound. When  $p = 1$ , run time always goes to  $O(n+n(n-1)/2)$ , as the upper bound. For other probabilities, this algorithm<sup>1</sup> runs in  $O(n+m)$  time.

For  $0 < p < 1$ , the algorithm is based on the implementation of the networkx function `fast_gnp_random_graph`<sup>2</sup>

### Parameters

- **num\_nodes** (*int*) – The number of nodes to create in the graph
- **probability** (*float*) – The probability of creating an edge between two nodes
- **seed** (*int*) – An optional seed to use for the random number generator

**Returns** A PyGraph object

**Return type** *PyGraph*

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## 2.3.3 networkx.directed\_gnm\_random\_graph

**directed\_gnm\_random\_graph**(*num\_nodes*, *num\_edges*, *seed=None, /*)

Return a  $G_{nm}$  of a directed graph

Generates a random directed graph out of all the possible graphs with  $n$  nodes and  $m$  edges. The generated graph will not be a multigraph and will not have self loops.

For  $n$  nodes, the maximum edges that can be returned is  $n(n-1)$ . Passing  $m$  higher than that will still return the maximum number of edges. If  $m = 0$ , the returned graph will always be empty (no edges). When a seed is provided, the results are reproducible. Passing a seed when  $m = 0$  or  $m \geq n(n-1)$  has no effect, as the result will always be an empty or a complete graph respectively.

This algorithm has a time complexity of  $O(n+m)$

### Parameters

- **num\_nodes** (*int*) – The number of nodes to create in the graph
- **num\_edges** (*int*) – The number of edges to create in the graph
- **seed** (*int*) – An optional seed to use for the random number generator

<sup>1</sup> Vladimir Batagelj and Ulrik Brandes, “Efficient generation of large random networks”, Phys. Rev. E, 71, 036113, 2005.

<sup>2</sup> [https://github.com/networkx/networkx/blob/networkx-2.4/networkx/generators/random\\_graphs.py#L49-L120](https://github.com/networkx/networkx/blob/networkx-2.4/networkx/generators/random_graphs.py#L49-L120)

**Returns** A PyDiGraph object

**Return type** *PyDiGraph*

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

### 2.3.4 networkx.undirected\_gnm\_random\_graph

**undirected\_gnm\_random\_graph**(*num\_nodes, probability, seed=None, /*)

Return a  $G_{nm}$  of an undirected graph

Generates a random undirected graph out of all the possible graphs with  $n$  nodes and  $m$  edges. The generated graph will not be a multigraph and will not have self loops.

For  $n$  nodes, the maximum edges that can be returned is  $n(n - 1)/2$ . Passing  $m$  higher than that will still return the maximum number of edges. If  $m = 0$ , the returned graph will always be empty (no edges). When a seed is provided, the results are reproducible. Passing a seed when  $m = 0$  or  $m \geq n(n - 1)/2$  has no effect, as the result will always be an empty or a complete graph respectively.

This algorithm has a time complexity of  $O(n + m)$

**Parameters**

- **num\_nodes** (*int*) – The number of nodes to create in the graph
- **num\_edges** (*int*) – The number of edges to create in the graph
- **seed** (*int*) – An optional seed to use for the random number generator

**Returns** A PyGraph object

**Return type** *PyGraph*

## 2.4 Algorithm Functions

### 2.4.1 Specific Graph Type Methods

<code>networkx.bfs_successors</code> (graph, node, /)	Return successors in a breadth-first-search from a source node.
<code>networkx.dag_longest_path</code> (graph, /)	Find the longest path in a DAG
<code>networkx.dag_longest_path_length</code> (graph, /)	Find the length of the longest path in a DAG
<code>networkx.number_weakly_connected_components</code> (graph, /)	Find the number of weakly connected components in a DAG.
<code>networkx.weakly_connected_components</code> (graph, /)	Find the weakly connected components in a directed graph
<code>networkx.is_weakly_connected</code> (graph, /)	Check if the graph is weakly connected
<code>networkx.is_directed_acyclic_graph</code> (graph, /)	Check that the PyDiGraph or PyDAG doesn't have a cycle
<code>networkx.is_isomorphic</code> (first, second, /)	Determine if 2 graphs are structurally isomorphic

continues on next page

Table 10 – continued from previous page

<code>networkx.is_isomorphic_node_match(first, ...)</code>	Determine if 2 DAGs are isomorphic
<code>networkx.topological_sort(graph, /)</code>	Return the topological sort of node indexes from the provided graph
<code>networkx.descendants(graph, node, /)</code>	Return the descendants of a node in a graph.
<code>networkx.ancestors(graph, node, /)</code>	Return the ancestors of a node in a graph.
<code>networkx.lexicographical_topological_sort(...)</code>	Get the lexicographical topological sorted nodes from the provided DAG
<code>networkx.graph_distance_matrix(graph, /[, ...])</code>	Get the distance matrix for an undirected graph
<code>networkx.digraph_distance_matrix(graph, /[, ...])</code>	Get the distance matrix for a directed graph
<code>networkx.floyd_warshall(dag, /)</code>	Return the shortest path lengths between ever pair of nodes that has a path connecting them
<code>networkx.graph_floyd_warshall_numpy(graph, /)</code>	Find all-pairs shortest path lengths using Floyd’s algorithm
<code>networkx.digraph_floyd_warshall_numpy</code>	Find all-pairs shortest path lengths using Floyd’s algorithm
<code>networkx.collect_runs(graph, filter)</code>	Collect runs that match a filter function
<code>networkx.layers(dag, first_layer, /)</code>	Return a list of layers
<code>networkx.digraph_adjacency_matrix(graph, /)</code>	Return the adjacency matrix for a PyDiGraph object
<code>networkx.graph_adjacency_matrix(graph, /[, ...])</code>	Return the adjacency matrix for a PyGraph class
<code>networkx.graph_all_simple_paths</code>	Return all simple paths between 2 nodes in a PyGraph object
<code>networkx.digraph_all_simple_paths</code>	Return all simple paths between 2 nodes in a PyDiGraph object
<code>networkx.graph_astar_shortest_path(graph, ...)</code>	Compute the A* shortest path for a PyGraph
<code>networkx.digraph_astar_shortest_path(graph, ...)</code>	Compute the A* shortest path for a PyDiGraph
<code>networkx.graph_dijkstra_shortest_paths</code>	Find the shortest path from a node
<code>networkx.digraph_dijkstra_shortest_paths</code>	Find the shortest path from a node
<code>networkx.graph_dijkstra_shortest_path_lengths</code>	Compute the lengths of the shortest paths for a PyGraph object using Dijkstra’s algorithm
<code>networkx.digraph_dijkstra_shortest_path_lengths</code>	Compute the lengths of the shortest paths for a PyDiGraph object using Dijkstra’s algorithm
<code>networkx.graph_k_shortest_path_lengths(...)</code>	Compute the length of the kth shortest path
<code>networkx.digraph_k_shortest_path_lengths(...)</code>	Compute the length of the kth shortest path
<code>networkx.graph_greedy_color(graph, /)</code>	Color a PyGraph using a largest_first strategy greedy graph coloring.
<code>networkx.cycle_basis(graph, /[, root])</code>	Return a list of cycles which form a basis for cycles of a given PyGraph
<code>networkx.strongly_connected_components(graph, /)</code>	Compute the strongly connected components for a directed graph
<code>networkx.graph_dfs_edges(graph, /[, source])</code>	Get edge list in depth first order
<code>networkx.digraph_dfs_edges(graph, /[, source])</code>	Get edge list in depth first order
<code>networkx.digraph_find_cycle(graph, /[, source])</code>	Return the first cycle encountered during DFS of a given PyDiGraph, empty list is returned if no cycle is found
<code>networkx.digraph_union(first, second, ...)</code>	Return a new PyDiGraph by forming a union from two input PyDiGraph objects
<code>networkx.is_matching(graph, matching, /)</code>	Check if matching is valid for graph
<code>networkx.is_maximal_matching(graph, matching, /)</code>	Check if a matching is a maximal ( <b>not</b> maximum) matching for a graph

continues on next page

Table 10 – continued from previous page

<code>retworkx.max_weight_matching</code> (graph, /[, ...])	Compute a maximum-weighted matching for a <i>PyGraph</i>
---	--

**Warning:** The documentation for retworkx has migrated to:

<https://qiskit.org/documentation/retworkx>

These docs will no longer be updated.

## retworkx.bfs\_successors

`bfs_successors`(graph, node, /)

Return successors in a breadth-first-search from a source node.

The return format is [(Parent Node, [Children Nodes])] in a bfs order from the source node provided.

### Parameters

- **graph** (*PyDiGraph*) – The DAG to get the `bfs_successors` from
- **node** (*int*) – The index of the dag node to get the bfs successors for

**Returns** A list of nodes’s data and their children in bfs order. The `BFSSuccessors` class that is returned is a custom container class that implements the sequence protocol. This can be used as a python list with index based access.

**Return type** *BFSSuccessors*

**Warning:** The documentation for retworkx has migrated to:

<https://qiskit.org/documentation/retworkx>

These docs will no longer be updated.

## retworkx.dag\_longest\_path

`dag_longest_path`(graph, /)

Find the longest path in a DAG

**Parameters** **graph** (*PyDiGraph*) – The graph to find the longest path on. The input object must be a DAG without a cycle.

**Returns** The node indices of the longest path on the DAG

**Return type** *NodeIndices*

### Raises

- **Exception** – If an unexpected error occurs or a path can’t be found
- **DAGHasCycle** – If the input *PyDiGraph* has a cycle

**Warning:** The documentation for retworkx has migrated to:

<https://qiskit.org/documentation/retworkx>

These docs will no longer be updated.

### networkx.dag\_longest\_path\_length

**dag\_longest\_path\_length**(*graph*, /)

Find the length of the longest path in a DAG

**Parameters** **graph** (*PyDiGraph*) – The graph to find the longest path on. The input object must be a DAG without a cycle.

**Returns** The longest path length on the DAG

**Return type** int

**Raises**

- **Exception** – If an unexpected error occurs or a path can't be found
- *DAGHasCycle* – If the input *PyDiGraph* has a cycle

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

### networkx.number\_weakly\_connected\_components

**number\_weakly\_connected\_components**(*graph*, /)

Find the number of weakly connected components in a DAG.

**Parameters** **graph** (*PyDiGraph*) – The graph to find the number of weakly connected components on

**Returns** The number of weakly connected components in the DAG

**Return type** int

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

### networkx.weakly\_connected\_components

**weakly\_connected\_components**(*graph*, /)

Find the weakly connected components in a directed graph

**Parameters** **graph** (*PyDiGraph*) – The graph to find the weakly connected components in

**Returns** A list of sets where each set is a weakly connected component of the graph

**Return type** list

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>



These docs will no longer be updated.

### retworkx.is\_weakly\_connected

**is\_weakly\_connected**(*graph*, /)

Check if the graph is weakly connected

**Parameters** **graph** (*PyDiGraph*) – The graph to check if it is weakly connected

**Returns** Whether the graph is weakly connected or not

**Return type** bool

**Raises** *NullGraph* – If an empty graph is passed in

**Warning:** The documentation for retworkx has migrated to:

<https://qiskit.org/documentation/retworkx>

These docs will no longer be updated.

### retworkx.is\_directed\_acyclic\_graph

**is\_directed\_acyclic\_graph**(*graph*, /)

Check that the PyDiGraph or PyDAG doesn't have a cycle

**Parameters** **graph** (*PyDiGraph*) – The graph to check for cycles

**Returns** True if there are no cycles in the input graph, False if there are cycles

**Return type** bool

**Warning:** The documentation for retworkx has migrated to:

<https://qiskit.org/documentation/retworkx>

These docs will no longer be updated.

### retworkx.is\_isomorphic

**is\_isomorphic**(*first*, *second*, /)

Determine if 2 graphs are structurally isomorphic

This checks if 2 graphs are structurally isomorphic (it doesn't match the contents of the nodes or edges on the graphs).

**Parameters**

- **first** (*PyDiGraph*) – The first graph to compare
- **second** (*PyDiGraph*) – The second graph to compare

**Returns** True if the 2 graphs are structurally isomorphic, False if they are not

**Return type** bool

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## networkx.is\_isomorphic\_node\_match

**is\_isomorphic\_node\_match**(*first, second, matcher, /*)

Determine if 2 DAGs are isomorphic

This checks if 2 graphs are isomorphic both structurally and also comparing the node data using the provided matcher function. The matcher function takes in 2 node data objects and will compare them. A simple example that checks if they're just equal would be:

```
graph_a = networkx.PyDAG()
graph_b = networkx.PyDAG()
networkx.is_isomorphic_node_match(graph_a, graph_b,
                                  lambda x, y: x == y)
```

### Parameters

- **first** (*PyDiGraph*) – The first graph to compare
- **second** (*PyDiGraph*) – The second graph to compare
- **matcher** (*callable*) – A python callable object that takes 2 positional one for each node data object. If the return of this function evaluates to True then the nodes passed to it are vieded as matching.

**Returns** True if the 2 graphs are isomorphic False if they are not.

**Return type** bool

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## networkx.topological\_sort

**topological\_sort**(*graph, /*)

Return the topological sort of node indexes from the provided graph

**Parameters** **graph** (*PyDiGraph*) – The DAG to get the topological sort on

**Returns** A list of node indices topologically sorted.

**Return type** *NodeIndices*

**Raises** *DAGHasCycle* – if a cycle is encountered while sorting the graph

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## networkx.descendants

**descendants**(*graph, node, /*)

Return the descendants of a node in a graph.

This differs from *PyDiGraph.successors()* method in that *successors* returns only nodes with a direct edge out of the provided node. While this function returns all nodes that have a path from the provided node.

### Parameters

- **graph** (*PyDiGraph*) – The graph to get the descendants from
- **node** (*int*) – The index of the graph node to get the descendants for

**Returns** A list of node indexes of descendants of provided node.

**Return type** list

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## networkx.ancestors

**ancestors**(*graph, node, /*)

Return the ancestors of a node in a graph.

This differs from *PyDiGraph.predecessors()* method in that *predecessors* returns only nodes with a direct edge into the provided node. While this function returns all nodes that have a path into the provided node.

### Parameters

- **graph** (*PyDiGraph*) – The graph to get the descendants from
- **node** (*int*) – The index of the graph node to get the ancestors for

**Returns** A list of node indexes of ancestors of provided node.

**Return type** list

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## networkx.lexicographical\_topological\_sort

### lexicographical\_topological\_sort(*dag*, *key*, /)

Get the lexicographical topological sorted nodes from the provided DAG

This function returns a list of nodes data in a graph lexicographically topologically sorted using the provided key function.

#### Parameters

- **dag** (*PyDiGraph*) – The DAG to get the topological sorted nodes from
- **key** (*callable*) – key is a python function or other callable that gets passed a single argument the node data from the graph and is expected to return a string which will be used for sorting.

**Returns** A list of node's data lexicographically topologically sorted.

**Return type** list

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## networkx.graph\_distance\_matrix

### graph\_distance\_matrix(*graph*, /, *parallel\_threshold*=300)

Get the distance matrix for an undirected graph

This differs from functions like `digraph_floyd_warshall_numpy` in that the edge weight/data payload is not used and each edge is treated as a distance of 1.

This function is also multithreaded and will run in parallel if the number of nodes in the graph is above the value of `parallel_threshold` (it defaults to 300). If the function will be running in parallel the env var `RAYON_NUM_THREADS` can be used to adjust how many threads will be used.

#### Parameters

- **graph** (*PyGraph*) – The graph to get the distance matrix for
- **parallel\_threshold** (*int*) – The number of nodes to calculate the the distance matrix in parallel at. It defaults to 300, but this can be tuned

**Returns** The distance matrix

**Return type** `numpy.ndarray`

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## networkx.digraph\_distance\_matrix

**digraph\_distance\_matrix**(*graph*, /, *parallel\_threshold*=300, *as\_undirected*=False)

Get the distance matrix for a directed graph

This differs from functions like `digraph_floyd_warshall_numpy` in that the edge weight/data payload is not used and each edge is treated as a distance of 1.

This function is also multithreaded and will run in parallel if the number of nodes in the graph is above the value of `parallel_threshold` (it defaults to 300). If the function will be running in parallel the env var `RAYON_NUM_THREADS` can be used to adjust how many threads will be used.

### Parameters

- **graph** (*PyDiGraph*) – The graph to get the distance matrix for
- **parallel\_threshold** (*int*) – The number of nodes to calculate the the distance matrix in parallel at. It defaults to 300, but this can be tuned
- **as\_undirected** (*bool*) – If set to True the input directed graph will be treat as if each edge was bidirectional/undirected in the output distance matrix.

**Returns** The distance matrix

**Return type** `numpy.ndarray`

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## networkx.floyd\_warshall

**floyd\_warshall**(*dag*, /)

Return the shortest path lengths between ever pair of nodes that has a path connecting them

The runtime is  $O(|N|^3 + |E|)$  where  $|N|$  is the number of nodes and  $|E|$  is the number of edges.

This is done with the Floyd Warshall algorithm:

1. Process all edges by setting the distance from the parent to the child equal to the edge weight.
2. Iterate through every pair of nodes (source, target) and an additional intermediary node (w). If the distance from source  $\rightarrow$  w  $\rightarrow$  target is less than the distance from source  $\rightarrow$  target, update the source  $\rightarrow$  target distance (to pass through w).

The return format is {Source Node: {Target Node: Distance}}.

---

**Note:** Paths that do not exist are simply not found in the return dictionary, rather than setting the distance to infinity, or -1.

---



---

**Note:** Edge weights are restricted to 1 in the current implementation.

---

**Parameters** **graph** (*PyDiGraph*) – The DiGraph to get all shortest paths from

**Returns** A dictionary of shortest paths

**Return type** dict

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## networkx.graph\_floyd\_warshall\_numpy

**graph\_floyd\_warshall\_numpy**(*graph*, /, *weight\_fn*=None, *default\_weight*=1.0)

Find all-pairs shortest path lengths using Floyd's algorithm

Floyd's algorithm is used for finding shortest paths in dense graphs or graphs with negative weights (where Dijkstra's algorithm fails).

### Parameters

- **graph** (*PyGraph*) – The graph to run Floyd's algorithm on
- **weight\_fn** – A callable object (function, lambda, etc) which will be passed the edge object and expected to return a float. This tells networkx/rust how to extract a numerical weight as a float for edge object. Some simple examples are:

```
graph_floyd_warshall_numpy(graph, weight_fn: lambda x: 1)
```

to return a weight of 1 for all edges. Also:

```
graph_floyd_warshall_numpy(graph, weight_fn: lambda x: float(x))
```

to cast the edge object as a float as the weight.

**Returns** A matrix of shortest path distances between nodes. If there is no path between two nodes then the corresponding matrix entry will be `np.inf`.

**Return type** numpy.ndarray

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## networkx.digraph\_floyd\_warshall\_numpy

**digraph\_floyd\_warshall\_numpy**()

Find all-pairs shortest path lengths using Floyd's algorithm

Floyd's algorithm is used for finding shortest paths in dense graphs or graphs with negative weights (where Dijkstra's algorithm fails).

### Parameters

- **graph** (*PyDiGraph*) – The directed graph to run Floyd's algorithm on

- **weight\_fn** – A callable object (function, lambda, etc) which will be passed the edge object and expected to return a float. This tells networkx/rust how to extract a numerical weight as a float for edge object. Some simple examples are:

```
graph_floyd_warshall_numpy(graph, weight_fn: lambda x: 1)
```

to return a weight of 1 for all edges. Also:

```
graph_floyd_warshall_numpy(graph, weight_fn: lambda x: float(x))
```

to cast the edge object as a float as the weight.

- **as\_undirected** – If set to true each directed edge will be treated as bidirectional/undirected.

**Returns** A matrix of shortest path distances between nodes. If there is no path between two nodes then the corresponding matrix entry will be `np.inf`.

**Return type** `numpy.ndarray`

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## networkx.collect\_runs

**collect\_runs**(*graph*, *filter*)

Collect runs that match a filter function

A run is a path of nodes where there is only a single successor and all nodes in the path match the given condition. Each node in the graph can appear in only a single run.

### Parameters

- **graph** (`PyDiGraph`) – The graph to find runs in
- **filter\_fn** – The filter function to use for matching nodes. It takes in one argument, the node data payload/weight object, and will return a boolean whether the node matches the conditions or not. If it returns `False` it will skip that node.

**Returns** a list of runs, where each run is a list of node data payload/weight for the nodes in the run

**Return type** `list`

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## networkx.layers

**layers**(*dag*, *first\_layer*, /)

Return a list of layers

A layer is a subgraph whose nodes are disjoint, i.e., a layer has depth 1. The layers are constructed using a greedy algorithm.

### Parameters

- **graph** (*PyDiGraph*) – The DAG to get the layers from
- **first\_layer** (*list*) – A list of node ids for the first layer. This will be the first layer in the output

**Returns** A list of layers, each layer is a list of node data

**Return type** list

**Raises** *InvalidNode* – If a node index in *first\_layer* is not in the graph

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## networkx.digraph\_adjacency\_matrix

**digraph\_adjacency\_matrix**(*graph*, /, *weight\_fn*=None, *default\_weight*=1.0)

Return the adjacency matrix for a PyDiGraph object

In the case where there are multiple edges between nodes the value in the output matrix will be the sum of the edges' weights.

### Parameters

- **graph** (*PyDiGraph*) – The DiGraph used to generate the adjacency matrix from
- **weight\_fn** (*callable*) – A callable object (function, lambda, etc) which will be passed the edge object and expected to return a float. This tells networkx/rust how to extract a numerical weight as a float for edge object. Some simple examples are:

```
dag_adjacency_matrix(dag, weight_fn: lambda x: 1)
```

to return a weight of 1 for all edges. Also:

```
dag_adjacency_matrix(dag, weight_fn: lambda x: float(x))
```

to cast the edge object as a float as the weight. If this is not specified a default value (either *default\_weight* or 1) will be used for all edges.

- **default\_weight** (*float*) –

**If *weight\_fn* is not used this can be** optionally used to specify a default weight to use for all edges.

**return** The adjacency matrix for the input dag as a numpy array

**rtype** numpy.ndarray



**Warning:** The documentation for retworkx has migrated to:

<https://qiskit.org/documentation/retworkx>

These docs will no longer be updated.

## retworkx.graph\_adjacency\_matrix

**graph\_adjacency\_matrix**(*graph*, /, *weight\_fn*=None, *default\_weight*=1.0)

Return the adjacency matrix for a PyGraph class

In the case where there are multiple edges between nodes the value in the output matrix will be the sum of the edges' weights.

### Parameters

- **graph** (*PyGraph*) – The graph used to generate the adjacency matrix from
- **weight\_fn** – A callable object (function, lambda, etc) which will be passed the edge object and expected to return a `float`. This tells retworkx/rust how to extract a numerical weight as a `float` for edge object. Some simple examples are:

```
graph_adjacency_matrix(graph, weight_fn: lambda x: 1)
```

to return a weight of 1 for all edges. Also:

```
graph_adjacency_matrix(graph, weight_fn: lambda x: float(x))
```

to cast the edge object as a float as the weight. If this is not specified a default value (either `default_weight` or 1) will be used for all edges.

- **default\_weight** (*float*) – If `weight_fn` is not used this can be optionally used to specify a default weight to use for all edges.

**Returns** The adjacency matrix for the input dag as a numpy array

**Return type** `numpy.ndarray`

**Warning:** The documentation for retworkx has migrated to:

<https://qiskit.org/documentation/retworkx>

These docs will no longer be updated.

## retworkx.graph\_all\_simple\_paths

**graph\_all\_simple\_paths**()

Return all simple paths between 2 nodes in a PyGraph object

A simple path is a path with no repeated nodes.

### Parameters

- **graph** (*PyGraph*) – The graph to find the path in
- **from** (*int*) – The node index to find the paths from
- **to** (*int*) – The node index to find the paths to

- **min\_depth** (*int*) – The minimum depth of the path to include in the output list of paths. By default all paths are included regardless of depth, setting to 0 will behave like the default.
- **cutoff** (*int*) – The maximum depth of path to include in the output list of paths. By default includes all paths regardless of depth, setting to 0 will behave like default.

**Returns** A list of lists where each inner list is a path of node indices

**Return type** list

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## networkx.digraph\_all\_simple\_paths

### digraph\_all\_simple\_paths()

Return all simple paths between 2 nodes in a PyDiGraph object

A simple path is a path with no repeated nodes.

#### Parameters

- **graph** (*PyDiGraph*) – The graph to find the path in
- **from** (*int*) – The node index to find the paths from
- **to** (*int*) – The node index to find the paths to
- **min\_depth** (*int*) – The minimum depth of the path to include in the output list of paths. By default all paths are included regardless of depth, sett to 0 will behave like the default.
- **cutoff** (*int*) – The maximum depth of path to include in the output list of paths. By default includes all paths regardless of depth, setting to 0 will behave like default.

**Returns** A list of lists where each inner list is a path

**Return type** list

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## networkx.graph\_astar\_shortest\_path

### graph\_astar\_shortest\_path(*graph, node, goal\_fn, edge\_cost, estimate\_cost, /*)

Compute the A\* shortest path for a PyGraph

#### Parameters

- **graph** (*PyGraph*) – The input graph to use
- **node** (*int*) – The node index to compute the path from
- **goal\_fn** – A python callable that will take in 1 parameter, a node's data object and will return a boolean which will be True if it is the finish node.

- **edge\_cost\_fn** – A python callable that will take in 1 parameter, an edge’s data object and will return a float that represents the cost of that edge. It must be non-negative.
- **estimate\_cost\_fn** – A python callable that will take in 1 parameter, a node’s data object and will return a float which represents the estimated cost for the next node. The return must be non-negative. For the algorithm to find the actual shortest path, it should be admissible, meaning that it should never overestimate the actual cost to get to the nearest goal node.

**Returns** The computed shortest path between node and finish as a list of node indices.

**Return type** *NodeIndices*

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## networkx.digraph\_astar\_shortest\_path

**digraph\_astar\_shortest\_path**(*graph, node, goal\_fn, edge\_cost, estimate\_cost, /*)

Compute the A\* shortest path for a PyDiGraph

### Parameters

- **graph** (*PyDiGraph*) – The input graph to use
- **node** (*int*) – The node index to compute the path from
- **goal\_fn** – A python callable that will take in 1 parameter, a node’s data object and will return a boolean which will be True if it is the finish node.
- **edge\_cost\_fn** – A python callable that will take in 1 parameter, an edge’s data object and will return a float that represents the cost of that edge. It must be non-negative.
- **estimate\_cost\_fn** – A python callable that will take in 1 parameter, a node’s data object and will return a float which represents the estimated cost for the next node. The return must be non-negative. For the algorithm to find the actual shortest path, it should be admissible, meaning that it should never overestimate the actual cost to get to the nearest goal node.

**Returns** The computed shortest path between node and finish as a list of node indices.

**Return type** *NodeIndices*

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## networkx.graph\_dijkstra\_shortest\_paths

### graph\_dijkstra\_shortest\_paths()

Find the shortest path from a node

This function will generate the shortest path from a source node using Dijkstra's algorithm.

#### Parameters

- **graph** (*PyGraph*) –
- **source** (*int*) – The node index to find paths from
- **target** (*int*) – An optional target to find a path to
- **weight\_fn** – An optional weight function for an edge. It will accept a single argument, the edge's weight object and will return a float which will be used to represent the weight/cost of the edge
- **default\_weight** (*float*) – If `weight_fn` isn't specified this optional float value will be used for the weight/cost of each edge.
- **as\_undirected** (*bool*) – If set to true the graph will be treated as undirected for finding the shortest path.

**Returns** Dictionary of paths. The keys are destination node indices and the dict values are lists of node indices making the path.

**Return type** dict

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## networkx.digraph\_dijkstra\_shortest\_paths

### digraph\_dijkstra\_shortest\_paths()

Find the shortest path from a node

This function will generate the shortest path from a source node using Dijkstra's algorithm.

#### Parameters

- **graph** (*PyDiGraph*) –
- **source** (*int*) – The node index to find paths from
- **target** (*int*) – An optional target path to find the path
- **weight\_fn** – An optional weight function for an edge. It will accept a single argument, the edge's weight object and will return a float which will be used to represent the weight/cost of the edge
- **default\_weight** (*float*) – If `weight_fn` isn't specified this optional float value will be used for the weight/cost of each edge.
- **as\_undirected** (*bool*) – If set to true the graph will be treated as undirected for finding the shortest path.

**Returns** Dictionary of paths. The keys are destination node indices and the dict values are lists of node indices making the path.

**Return type** dict

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## networkx.graph\_dijkstra\_shortest\_path\_lengths

**graph\_dijkstra\_shortest\_path\_lengths**(*graph*, *node*, *edge\_cost\_fn*, /, *goal=None*)

Compute the lengths of the shortest paths for a PyGraph object using Dijkstra's algorithm

### Parameters

- **graph** (*PyGraph*) – The input graph to use
- **node** (*int*) – The node index to use as the source for finding the shortest paths from
- **edge\_cost\_fn** – A python callable that will take in 1 parameter, an edge's data object and will return a float that represents the cost/weight of that edge. It must be non-negative
- **goal** (*int*) – An optional node index to use as the end of the path. When specified the traversal will stop when the goal is reached and the output dictionary will only have a single entry with the length of the shortest path to the goal node.

**Returns** A dictionary of the shortest paths from the provided node where the key is the node index of the end of the path and the value is the cost/sum of the weights of path

**Return type** dict

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## networkx.digraph\_dijkstra\_shortest\_path\_lengths

**digraph\_dijkstra\_shortest\_path\_lengths**(*graph*, *node*, *edge\_cost\_fn*, /, *goal=None*)

Compute the lengths of the shortest paths for a PyDiGraph object using Dijkstra's algorithm

### Parameters

- **graph** (*PyDiGraph*) – The input graph to use
- **node** (*int*) – The node index to use as the source for finding the shortest paths from
- **edge\_cost\_fn** – A python callable that will take in 1 parameter, an edge's data object and will return a float that represents the cost/weight of that edge. It must be non-negative
- **goal** (*int*) – An optional node index to use as the end of the path. When specified the traversal will stop when the goal is reached and the output dictionary will only have a single entry with the length of the shortest path to the goal node.

**Returns** A dictionary of the shortest paths from the provided node where the key is the node index of the end of the path and the value is the cost/sum of the weights of path

**Return type** dict

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## networkx.graph\_k\_shortest\_path\_lengths

**graph\_k\_shortest\_path\_lengths**(*graph*, *start*, *k*, *edge\_cost*, *l*, *goal=None*)

Compute the length of the kth shortest path

Computes the lengths of the kth shortest path from *start* to every reachable node.

Computes in  $O(k * (|E| + |V| * \log(|V|)))$  time (average).

### Parameters

- **graph** (*PyGraph*) – The graph to find the shortest paths in
- **start** (*int*) – The node index to find the shortest paths from
- **k** (*int*) – The kth shortest path to find the lengths of
- **edge\_cost** – A python callable that will receive an edge payload and return a float for the cost of that eedge
- **goal** (*int*) – An optional goal node index, if specified the output dictionary

**Returns** A dict of lengths where the key is the destination node index and the value is the length of the path.

**Return type** dict

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## networkx.digraph\_k\_shortest\_path\_lengths

**digraph\_k\_shortest\_path\_lengths**(*graph*, *start*, *k*, *edge\_cost*, *l*, *goal=None*)

Compute the length of the kth shortest path

Computes the lengths of the kth shortest path from *start* to every reachable node.

Computes in  $O(k * (|E| + |V| * \log(|V|)))$  time (average).

### Parameters

- **graph** (*PyGraph*) – The graph to find the shortest paths in
- **start** (*int*) – The node index to find the shortest paths from
- **k** (*int*) – The kth shortest path to find the lengths of

- **edge\_cost** – A python callable that will receive an edge payload and return a float for the cost of that eedge
- **goal** (*int*) – An optional goal node index, if specified the output dictionary

**Returns** A dict of lengths where the key is the destination node index and the value is the length of the path.

**Return type** dict

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## networkx.graph\_greedy\_color

**graph\_greedy\_color**(*graph*, /)

Color a PyGraph using a largest\_first strategy greedy graph coloring.

**Parameters** **PyGraph** – The input PyGraph object to color

**Returns** A dictionary where keys are node indices and the value is the color

**Return type** dict

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## networkx.cycle\_basis

**cycle\_basis**(*graph*, /, *root=None*)

Return a list of cycles which form a basis for cycles of a given PyGraph

A basis for cycles of a graph is a minimal collection of cycles such that any cycle in the graph can be written as a sum of cycles in the basis. Here summation of cycles is defined as the exclusive or of the edges.

This is adapted from algorithm CACM 491<sup>1</sup>.

**Parameters**

- **graph** (**PyGraph**) – The graph to find the cycle basis in
- **root** (*int*) – Optional index for starting node for basis

**Returns** A list of cycle lists. Each list is a list of node ids which forms a cycle (loop) in the input graph

**Return type** list

<sup>1</sup> Paton, K. An algorithm for finding a fundamental set of cycles of a graph. Comm. ACM 12, 9 (Sept 1969), 514-518.

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

### networkx.strongly\_connected\_components

**strongly\_connected\_components**(*graph*, /)

Compute the strongly connected components for a directed graph

This function is implemented using Kosaraju's algorithm

**Parameters** **graph** (*PyDiGraph*) – The input graph to find the strongly connected components for.

**Returns** A list of list of node ids for strongly connected components

**Return type** list

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

### networkx.graph\_dfs\_edges

**graph\_dfs\_edges**(*graph*, /, *source=None*)

Get edge list in depth first order

**Parameters**

- **graph** (*PyGraph*) – The graph to get the DFS edge list from
- **source** (*int*) – An optional node index to use as the starting node for the depth-first search. The edge list will only return edges in the components reachable from this index. If this is not specified then a source will be chosen arbitrarily and repeated until all components of the graph are searched.

**Returns** A list of edges as a tuple of the form (*source*, *target*) in depth-first order

**Return type** *EdgeList*

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.



## networkx.digraph\_dfs\_edges

**digraph\_dfs\_edges**(*graph*, /, *source=None*)

Get edge list in depth first order

### Parameters

- **graph** (*PyDiGraph*) – The graph to get the DFS edge list from
- **source** (*int*) – An optional node index to use as the starting node for the depth-first search. The edge list will only return edges in the components reachable from this index. If this is not specified then a source will be chosen arbitrarily and repeated until all components of the graph are searched.

**Returns** A list of edges as a tuple of the form (*source*, *target*) in depth-first order

**Return type** *EdgeList*

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## networkx.digraph\_find\_cycle

**digraph\_find\_cycle**(*graph*, /, *source=None*)

Return the first cycle encountered during DFS of a given PyDiGraph, empty list is returned if no cycle is found

### Parameters

- **graph** (*PyDiGraph*) – The graph to find the cycle in
- **source** (*int*) – Optional index to find a cycle for. If not specified an arbitrary node will be selected from the graph.

**Returns** A list describing the cycle. The index of node ids which forms a cycle (loop) in the input graph

**Return type** *EdgeList*

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## networkx.digraph\_union

**digraph\_union**(*first, second, merge\_nodes, merge\_edges, /*)

Return a new PyDiGraph by forming a union from two input PyDiGraph objects

The algorithm in this function operates in three phases:

1. Add all the nodes from *second* into *first*. operates in  $O(n)$ , with  $n$  being number of nodes in *b*.
2. Merge nodes from *second* over *first* given that:
  - The *merge\_nodes* is *True*. operates in  $O(n^2)$ , with  $n$  being the number of nodes in *second*.
  - The respective node in *second* and *first* share the same weight/data payload.
3. Adds all the edges from *second* to *first*. If the *merge\_edges* parameter is *True* and the respective edge in *second* and *first* share the same weight/data payload they will be merged together.

**param PyDiGraph first** The first directed graph object

**param PyDiGraph second** The second directed graph object

**param bool merge\_nodes** If set to *True* nodes will be merged between *second* and *first* if the weights are equal.

**param bool merge\_edges** If set to *True* edges will be merged between *second* and *first* if the weights are equal.

**returns** A new PyDiGraph object that is the union of *second* and *first*. It's worth noting the weight/data payload objects are passed by reference from *first* and *second* to this new object.

**rtype** PyDiGraph

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## networkx.is\_matching

**is\_matching**(*graph, matching, /*)

Check if matching is valid for graph

A *matching* in a graph is a set of edges in which no two distinct edges share a common endpoint.

### Parameters

- **graph** (*PyDiGraph*) – The graph to check if the matching is valid for
- **matching** (*set*) – A set of node index tuples for each edge in the matching.

**Returns** Whether the provided matching is a valid matching for the graph

**Return type** bool

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## networkx.is\_maximal\_matching

**is\_maximal\_matching**(*graph*, *matching*, /)

Check if a matching is a maximal (**not** maximum) matching for a graph

A *maximal matching* in a graph is a matching in which adding any edge would cause the set to no longer be a valid matching.

---

**Note:** This is not checking for a *maximum* (globally optimal) matching, but a *maximal* (locally optimal) matching.

---

### Parameters

- **graph** (*PyDiGraph*) – The graph to check if the matching is maximal for.
- **matching** (*set*) – A set of node index tuples for each edge in the matching.

**Returns** Whether the provided matching is a valid matching and whether it is maximal or not.

**Return type** bool

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## networkx.max\_weight\_matching

**max\_weight\_matching**(*graph*, /, *max\_cardinality=False*, *weight\_fn=None*, *default\_weight=1*, *verify\_optimum=False*)

Compute a maximum-weighted matching for a *PyGraph*

A matching is a subset of edges in which no node occurs more than once. The weight of a matching is the sum of the weights of its edges. A maximal matching cannot add more edges and still be a matching. The cardinality of a matching is the number of matched edges.

This function takes time  $O(n^3)$  where  $n$  is the number of nodes in the graph.

This method is based on the “blossom” method for finding augmenting paths and the “primal-dual” method for finding a matching of maximum weight, both methods invented by Jack Edmonds<sup>1</sup>.

### Parameters

- **graph** (*PyGraph*) – The undirected graph to compute the max weight matching for. Expects to have no parallel edges (multigraphs are untested currently).

<sup>1</sup> “Efficient Algorithms for Finding Maximum Matching in Graphs”, Zvi Galil, ACM Computing Surveys, 1986.

- **max\_cardinality** (*bool*) – If True, compute the maximum-cardinality matching with maximum weight among all maximum-cardinality matchings. Defaults False.
- **weight\_fn** (*callable*) – An optional callable that will be passed a single argument the edge object for each edge in the graph. It is expected to return an `int` weight for that edge. For example, if the weights are all integers you can use: `lambda x: x`. If not specified the value for `default_weight` will be used for all edge weights.
- **default\_weight** (*int*) – The `int` value to use for all edge weights in the graph if `weight_fn` is not specified. Defaults to 1.
- **verify\_optimum** (*bool*) – A boolean flag to run a check that the found solution is optimum. If set to true an exception will be raised if the found solution is not optimum. This is mostly useful for testing.

**Returns** A set of tuples of the matching, Note that only a single direction will be listed in the output, for example: `{(0, 1), }`.

**Return type** `set`

## 2.4.2 Universal Functions

These functions are algorithm functions that wrap per graph object type functions in the algorithms API but can be run with a `PyGraph`, `PyDiGraph`, or `PyDAG` object.

<code>networkx.distance_matrix()</code>	Get the distance matrix for a graph
<code>networkx.floyd_warshall_numpy()</code>	Return the adjacency matrix for a graph object
<code>networkx.adjacency_matrix()</code>	Return the adjacency matrix for a graph object
<code>networkx.all_simple_paths()</code>	Return all simple paths between 2 nodes in a <code>PyGraph</code> object
<code>networkx.astar_shortest_path()</code>	Compute the A* shortest path for a graph
<code>networkx.dijkstra_shortest_paths()</code>	Find the shortest path from a node
<code>networkx.dijkstra_shortest_path_lengths()</code>	Compute the lengths of the shortest paths for a graph object using Dijkstra's algorithm.
<code>networkx.k_shortest_path_lengths()</code>	Compute the length of the kth shortest path
<code>networkx.dfs_edges()</code>	Get edge list in depth first order

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

### networkx.distance\_matrix

`distance_matrix(graph, parallel_threshold=300)`

`distance_matrix(graph: networkx.PyDiGraph, parallel_threshold=300, as_undirected=False)`

`distance_matrix(graph: networkx.PyGraph, parallel_threshold=300)`

Get the distance matrix for a graph

This differs from functions like `floyd_warshall_numpy()` in that the edge weight/data payload is not used and each edge is treated as a distance of 1.

This function is also multithreaded and will run in parallel if the number of nodes in the graph is above the

value of `parallel_threshold` (it defaults to 300). If the function will be running in parallel the env var `RAYON_NUM_THREADS` can be used to adjust how many threads will be used.

**Parameters**

- **graph** – The graph to get the distance matrix for, can be either a *PyGraph* or *PyDiGraph*.
- **parallel\_threshold** (*int*) – The number of nodes to calculate the the distance matrix in parallel at. It defaults to 300, but this can be tuned
- **as\_undirected** (*bool*) – If set to True the input directed graph will be treat as if each edge was bidirectional/undirected in the output distance matrix.

**Returns** The distance matrix

**Return type** `numpy.ndarray`

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

**networkx.floyd\_warshall\_numpy**

`floyd_warshall_numpy(graph, weight_fn=None, default_weight=1.0)`

`floyd_warshall_numpy(graph: networkx.PyDiGraph, weight_fn=None, default_weight=1.0)`

`floyd_warshall_numpy(graph: networkx.PyGraph, weight_fn=None, default_weight=1.0)`

Return the adjacency matrix for a graph object

In the case where there are multiple edges between nodes the value in the output matrix will be the sum of the edges' weights.

**Parameters**

- **graph** – The graph used to generate the adjacency matrix from. Can either be a *PyGraph* or *PyDiGraph*
- **weight\_fn** (*callable*) – A callable object (function, lambda, etc) which will be passed the edge object and expected to return a `float`. This tells networkx/rust how to extract a numerical weight as a `float` for edge object. Some simple examples are:

```
adjacency_matrix(graph, weight_fn: lambda x: 1)
```

to return a weight of 1 for all edges. Also:

```
adjacency_matrix(graph, weight_fn: lambda x: float(x))
```

to cast the edge object as a float as the weight. If this is not specified a default value (either `default_weight` or 1) will be used for all edges.

- **default\_weight** (*float*) –

If **weight\_fn** is not used this can be optionally used to specify a default weight to use for all edges.

**return** The adjacency matrix for the input dag as a numpy array

**rtype** `numpy.ndarray`

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## networkx.adjacency\_matrix

**adjacency\_matrix**(*graph*, *weight\_fn=None*, *default\_weight=1.0*)

**adjacency\_matrix**(*graph*: `networkx.PyDiGraph`, *weight\_fn=None*, *default\_weight=1.0*)

**adjacency\_matrix**(*graph*: `networkx.PyGraph`, *weight\_fn=None*, *default\_weight=1.0*)

Return the adjacency matrix for a graph object

In the case where there are multiple edges between nodes the value in the output matrix will be the sum of the edges' weights.

### Parameters

- **graph** – The graph used to generate the adjacency matrix from. Can either be a `PyGraph` or `PyDiGraph`
- **weight\_fn** (*callable*) – A callable object (function, lambda, etc) which will be passed the edge object and expected to return a `float`. This tells networkx/rust how to extract a numerical weight as a `float` for edge object. Some simple examples are:

```
adjacency_matrix(graph, weight_fn: lambda x: 1)
```

to return a weight of 1 for all edges. Also:

```
adjacency_matrix(graph, weight_fn: lambda x: float(x))
```

to cast the edge object as a float as the weight. If this is not specified a default value (either `default_weight` or 1) will be used for all edges.

- **default\_weight** (*float*) –

If **weight\_fn** is not used this can be optionally used to specify a default weight to use for all edges.

**return** The adjacency matrix for the input dag as a numpy array

**rtype** `numpy.ndarray`

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## networkx.all\_simple\_paths

`all_simple_paths(graph, from_, to, min_depth=None, cutoff=None)`

`all_simple_paths(graph: networkx.PyDiGraph, from_, to, min_depth=None, cutoff=None)`

`all_simple_paths(graph: networkx.PyGraph, from_, to, min_depth=None, cutoff=None)`

Return all simple paths between 2 nodes in a PyGraph object

A simple path is a path with no repeated nodes.

### Parameters

- **graph** – The graph to find the path in. Can either be a class:~*networkx.PyGraph* or *PyDiGraph*
- **from** (*int*) – The node index to find the paths from
- **to** (*int*) – The node index to find the paths to
- **min\_depth** (*int*) – The minimum depth of the path to include in the output list of paths. By default all paths are included regardless of depth, setting to 0 will behave like the default.
- **cutoff** (*int*) – The maximum depth of path to include in the output list of paths. By default includes all paths regardless of depth, setting to 0 will behave like default.

**Returns** A list of lists where each inner list is a path of node indices

**Return type** list

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## networkx.astar\_shortest\_path

`astar_shortest_path(graph, node, goal_fn, edge_cost_fn, estimate_cost_fn)`

`astar_shortest_path(graph: networkx.PyDiGraph, node, goal_fn, edge_cost_fn, estimate_cost_fn)`

`astar_shortest_path(graph: networkx.PyGraph, node, goal_fn, edge_cost_fn, estimate_cost_fn)`

Compute the A\* shortest path for a graph

### Parameters

- **graph** – The input graph to use. Can either be a *PyGraph* or *PyDiGraph*
- **node** (*int*) – The node index to compute the path from
- **goal\_fn** – A python callable that will take in 1 parameter, a node's data object and will return a boolean which will be True if it is the finish node.
- **edge\_cost\_fn** – A python callable that will take in 1 parameter, an edge's data object and will return a float that represents the cost of that edge. It must be non-negative.
- **estimate\_cost\_fn** – A python callable that will take in 1 parameter, a node's data object and will return a float which represents the estimated cost for the next node. The return must be non-negative. For the algorithm to find the actual shortest path, it should be admissible, meaning that it should never overestimate the actual cost to get to the nearest goal node.

**Returns** The computed shortest path between node and finish as a list of node indices.

**Return type** *NodeIndices*

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

### networkx.dijkstra\_shortest\_paths

**dijkstra\_shortest\_paths**(*graph*, *source*, *target=None*, *weight\_fn=None*, *default\_weight=1.0*, *as\_undirected=False*)

**dijkstra\_shortest\_paths**(*graph*: `networkx.PyDiGraph`, *source*, *target=None*, *weight\_fn=None*, *default\_weight=1.0*, *as\_undirected=False*)

**dijkstra\_shortest\_paths**(*graph*: `networkx.PyGraph`, *source*, *target=None*, *weight\_fn=None*, *default\_weight=1.0*)

Find the shortest path from a node

This function will generate the shortest path from a source node using Dijkstra's algorithm.

#### Parameters

- **graph** – The input graph to use. Can either be a `PyGraph` or `PyDiGraph`
- **source** (*int*) – The node index to find paths from
- **target** (*int*) – An optional target to find a path to
- **weight\_fn** – An optional weight function for an edge. It will accept a single argument, the edge's weight object and will return a float which will be used to represent the weight/cost of the edge
- **default\_weight** (*float*) – If `weight_fn` isn't specified this optional float value will be used for the weight/cost of each edge.
- **as\_undirected** (*bool*) – If set to true the graph will be treated as undirected for finding the shortest path. This only works with a `PyDiGraph` input for `graph`

**Returns** Dictionary of paths. The keys are destination node indices and the dict values are lists of node indices making the path.

**Return type** dict

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.



## networkx.dijkstra\_shortest\_path\_lengths

**dijkstra\_shortest\_path\_lengths**(*graph*, *node*, *edge\_cost\_fn*, *goal=None*)

**dijkstra\_shortest\_path\_lengths**(*graph*: `networkx.PyDiGraph`, *node*, *edge\_cost\_fn*, *goal=None*)

**dijkstra\_shortest\_path\_lengths**(*graph*: `networkx.PyGraph`, *node*, *edge\_cost\_fn*, *goal=None*)

Compute the lengths of the shortest paths for a graph object using Dijkstra's algorithm.

### Parameters

- **graph** – The input graph to use. Can either be a `PyGraph` or `PyDiGraph`
- **node** (*int*) – The node index to use as the source for finding the shortest paths from
- **edge\_cost\_fn** – A python callable that will take in 1 parameter, an edge's data object and will return a float that represents the cost/weight of that edge. It must be non-negative
- **goal** (*int*) – An optional node index to use as the end of the path. When specified the traversal will stop when the goal is reached and the output dictionary will only have a single entry with the length of the shortest path to the goal node.

**Returns** A dictionary of the shortest paths from the provided node where the key is the node index of the end of the path and the value is the cost/sum of the weights of path

**Return type** dict

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## networkx.k\_shortest\_path\_lengths

**k\_shortest\_path\_lengths**(*graph*, *start*, *k*, *edge\_cost*, *goal=None*)

**k\_shortest\_path\_lengths**(*graph*: `networkx.PyDiGraph`, *start*, *k*, *edge\_cost*, *goal=None*)

**k\_shortest\_path\_lengths**(*graph*: `networkx.PyGraph`, *start*, *k*, *edge\_cost*, *goal=None*)

Compute the length of the kth shortest path

Computes the lengths of the kth shortest path from *start* to every reachable node.

Computes in  $O(k * (|E| + |V| * \log(|V|)))$  time (average).

### Parameters

- **graph** – The graph to find the shortest paths in. Can either be a `PyGraph` or `PyDiGraph`
- **start** (*int*) – The node index to find the shortest paths from
- **k** (*int*) – The kth shortest path to find the lengths of
- **edge\_cost** – A python callable that will receive an edge payload and return a float for the cost of that eedge
- **goal** (*int*) – An optional goal node index, if specified the output dictionary

**Returns** A dict of lengths where the key is the destination node index and the value is the length of the path.

**Return type** dict

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## networkx.dfs\_edges

`dfs_edges(graph, source)`

`dfs_edges(graph: networkx.PyDiGraph, source)`

`dfs_edges(graph: networkx.PyGraph, source)`

Get edge list in depth first order

### Parameters

- **graph** (`PyGraph`) – The graph to get the DFS edge list from
- **source** (`int`) – An optional node index to use as the starting node for the depth-first search. The edge list will only return edges in the components reachable from this index. If this is not specified then a source will be chosen arbitrarily and repeated until all components of the graph are searched.

**Returns** A list of edges as a tuple of the form (source, target) in depth-first order

**Return type** EdgeList raise `TypeError("Invalid Input Type %s for graph" % type(graph))`

## 2.5 Exceptions

---

`networkx.InvalidNode`

---

`networkx.DAGWouldCycle`

---

`networkx.NoEdgeBetweenNodes`

---

`networkx.DAGHasCycle`

---

`networkx.NoSuitableNeighbors`

---

`networkx.NoPathFound`

---

`networkx.NullGraph`

---

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

### 2.5.1 retworkx.InvalidNode

exception InvalidNode

`with_traceback()`

Exception.with\_traceback(tb) – set self.\_\_traceback\_\_ to tb and return self.

**Warning:** The documentation for retworkx has migrated to:

<https://qiskit.org/documentation/retworkx>

These docs will no longer be updated.

### 2.5.2 retworkx.DAGWouldCycle

exception DAGWouldCycle

`with_traceback()`

Exception.with\_traceback(tb) – set self.\_\_traceback\_\_ to tb and return self.

**Warning:** The documentation for retworkx has migrated to:

<https://qiskit.org/documentation/retworkx>

These docs will no longer be updated.

### 2.5.3 retworkx.NoEdgeBetweenNodes

exception NoEdgeBetweenNodes

`with_traceback()`

Exception.with\_traceback(tb) – set self.\_\_traceback\_\_ to tb and return self.

**Warning:** The documentation for retworkx has migrated to:

<https://qiskit.org/documentation/retworkx>

These docs will no longer be updated.

## 2.5.4 networkx.DAGHasCycle

**exception** DAGHasCycle

**with\_traceback()**

Exception.with\_traceback(tb) – set self.\_\_traceback\_\_ to tb and return self.

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## 2.5.5 networkx.NoSuitableNeighbors

**exception** NoSuitableNeighbors

**with\_traceback()**

Exception.with\_traceback(tb) – set self.\_\_traceback\_\_ to tb and return self.

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## 2.5.6 networkx.NoPathFound

**exception** NoPathFound

**with\_traceback()**

Exception.with\_traceback(tb) – set self.\_\_traceback\_\_ to tb and return self.

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## 2.5.7 retworkx.NullGraph

### exception NullGraph

#### with\_traceback()

Exception.with\_traceback(tb) – set self.\_\_traceback\_\_ to tb and return self.

## 2.6 Return Iterator Types

<i>retworkx.BFSSuccessors</i>	A custom class for the return from <i>retworkx.bfs_successors()</i>
<i>retworkx.NodeIndices</i>	A custom class for the return of node indices
<i>retworkx.EdgeList</i>	A custom class for the return of edge lists
<i>retworkx.WeightedEdgeList</i>	A custom class for the return of edge lists with weights

**Warning:** The documentation for retworkx has migrated to:

<https://qiskit.org/documentation/retworkx>

These docs will no longer be updated.

### 2.6.1 retworkx.BFSSuccessors

#### class BFSSuccessors

A custom class for the return from *retworkx.bfs\_successors()*

This class is a container class for the results of the *retworkx.bfs\_successors()* function. It implements the Python sequence protocol. So you can treat the return as read-only sequence/list that is integer indexed. If you want to use it as an iterator you can by wrapping it in an *iter()* that will yield the results in order.

For example:

```
import retworkx

graph = retworkx.generators.directed_path_graph(5)
bfs_succ = retworkx.bfs_successors(0)
# Index based access
third_element = bfs_succ[2]
# Use as iterator
bfs_iter = iter(bfs_succ)
first_element = next(bfs_iter)
second_element = nex(bfs_iter)
```

#### \_\_init\_\_()

Initialize self. See help(type(self)) for accurate signature.

## Methods

---

<code>__init__()</code>	Initialize self.
-------------------------	------------------

---

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

## 2.6.2 networkx.NodeIndices

### class NodeIndices

A custom class for the return of node indices

This class is a container class for the results of functions that return a list of node indices. It implements the Python sequence protocol. So you can treat the return as a read-only sequence/list that is integer indexed. If you want to use it as an iterator you can by wrapping it in an `iter()` that will yield the results in order.

For example:

```
import networkx

graph = networkx.generators.directed_path_graph(5)
nodes = networkx.node_indexes(0)
# Index based access
third_element = nodes[2]
# Use as iterator
nodes_iter = iter(nodes)
first_element = next(nodes_iter)
second_element = next(nodes_iter)
```

`__init__()`  
Initialize self. See `help(type(self))` for accurate signature.

## Methods

---

<code>__init__()</code>	Initialize self.
-------------------------	------------------

---

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

### 2.6.3 networkx.EdgeList

#### class EdgeList

A custom class for the return of edge lists

This class is a container class for the results of functions that return a list of edges. It implements the Python sequence protocol. So you can treat the return as a read-only sequence/list that is integer indexed. If you want to use it as an iterator you can by wrapping it in an `iter()` that will yield the results in order.

For example:

```
import networkx

graph = networkx.generators.directed_path_graph(5)
edges = graph.edge_list()
# Index based access
third_element = edges[2]
# Use as iterator
edges_iter = iter(edges)
first_element = next(edges_iter)
second_element = next(edges_iter)
```

#### `__init__()`

Initialize self. See help(type(self)) for accurate signature.

#### Methods

---

#### `__init__()`

Initialize self.

---

**Warning:** The documentation for networkx has migrated to:

<https://qiskit.org/documentation/networkx>

These docs will no longer be updated.

### 2.6.4 networkx.WeightedEdgeList

#### class WeightedEdgeList

A custom class for the return of edge lists with weights

This class is a container class for the results of functions that return a list of edges with weights. It implements the Python sequence protocol. So you can treat the return as a read-only sequence/list that is integer indexed. If you want to use it as an iterator you can by wrapping it in an `iter()` that will yield the results in order.

For example:

```
import networkx

graph = networkx.generators.directed_path_graph(5)
edges = graph.weighted_edge_list()
# Index based access
third_element = edges[2]
```

(continues on next page)

(continued from previous page)

```
# Use as iterator
edges_iter = iter(edges)
first_element = next(edges_iter)
second_element = next(edges_iter)
```

**\_\_init\_\_()**

Initialize self. See help(type(self)) for accurate signature.

## Methods

---

**\_\_init\_\_()**

---

Initialize self.

---

**Warning:** The documentation for retworkx has migrated to:<https://qiskit.org/documentation/retworkx>

These docs will no longer be updated.



## RELEASE NOTES

### 3.1 0.8.0

#### 3.1.1 Prelude

This release includes several new features and bug fixes. The main features for this release are some usability improvements including the introduction of new methods for interacting with edges, constructing graphs from adjacency matrices, and *Universal Functions* that are not strictly typed and will work with either a *PyGraph* or *PyDiGraph* object. It also includes new algorithm functions around matchings for a *PyGraph*, including a function to find the maximum weight matching. This is also the first release to include support and publishing of precompiled binaries for Apple Arm CPUs on MacOS.

#### 3.1.2 New Features

- A new constructor method *from\_adjacency\_matrix()* has been added to the *PyDiGraph* and *PyGraph* (*from\_adjacency\_matrix()*) classes. It enables creating a new graph from an input adjacency\_matrix. For example:

```
import os
import tempfile

import numpy as np
import pydot
from PIL import Image

import networkx

# Adjacency matrix for directed outward star graph:
adjacency_matrix = np.array([
    [0., 1., 1., 1., 1.],
    [0., 0., 0., 0., 0.],
    [0., 0., 0., 0., 0.],
    [0., 0., 0., 0., 0.],
    [0., 0., 0., 0., 0.]])
# Create a graph from the adjacency_matrix:
graph = networkx.PyDiGraph.from_adjacency_matrix(adjacency_matrix)
# Draw graph
dot_str = graph.to_dot()
```

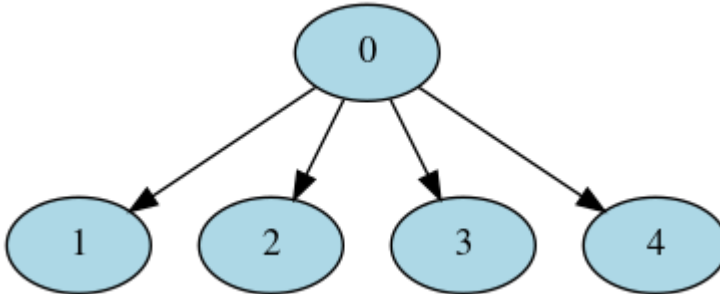
(continues on next page)

(continued from previous page)

```

lambda node: dict(
    color='black', fillcolor='lightblue', style='filled'))
dot = pydot.graph_from_dot_data(dot_str)[0]
with tempfile.TemporaryDirectory() as tmpdirname:
    tmp_path = os.path.join(tmpdirname, 'dag.png')
    dot.write_png(tmp_path)
    image = Image.open(tmp_path)
    os.remove(tmp_path)
image

```



- A new algorithm function, `is_matching()`, was added to check if a matching set is valid for given `PyGraph` object.
- A new algorithm function, `is_maximal_matching()`, was added to check if a matching set is valid and maximal for a given `PyGraph` object.
- Add a new function, `max_weight_matching()` for computing the maximum-weighted matching for a `PyGraph` object.
- The `PyGraph` and `PyDiGraph` constructors now have a new kwarg `multigraph` which can optionally be set to `False` to disallow adding parallel edges to the graph. With `multigraph=False` if an edge is attempted to be added where one already exists it will update the weight for the edge with the new value. For example:

```

import os
import tempfile

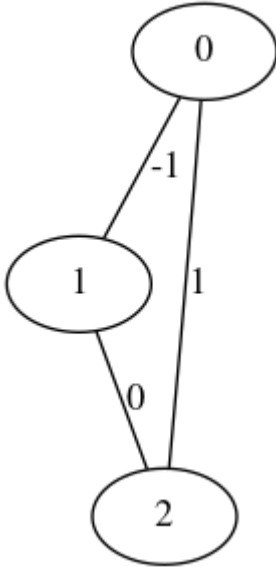
import pydot
from PIL import Image

import networkx as nx

graph = nx.Graph(multigraph=False)
graph.extend_from_weighted_edge_list([(0, 1, -1), (1, 2, 0), (2, 0, 1)])
dot = pydot.graph_from_dot_data(
    graph.to_dot(edge_attr=lambda e: {'label': str(e)})))[0]

with tempfile.TemporaryDirectory() as tmpdirname:
    tmp_path = os.path.join(tmpdirname, 'dag.png')
    dot.write_png(tmp_path)
    image = Image.open(tmp_path)
    os.remove(tmp_path)
image

```

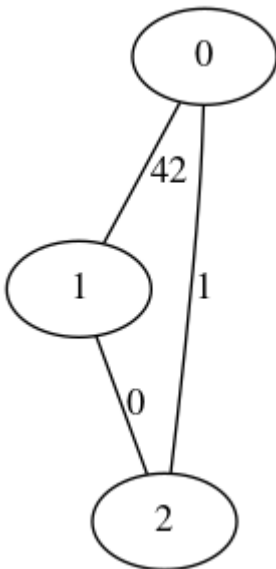


Then trying to add an edge between 0 and 1 again will update its weight/payload.

```

graph.add_edge(0, 1, 42)
dot = pydot.graph_from_dot_data(
    graph.to_dot(edge_attr=lambda e: {'label': str(e)})))[0]

with tempfile.TemporaryDirectory() as tmpdirname:
    tmp_path = os.path.join(tmpdirname, 'dag.png')
    dot.write_png(tmp_path)
    image = Image.open(tmp_path)
    os.remove(tmp_path)
image
  
```



You can query whether a PyGraph allows multigraphs with the boolean attribute `multigraph`. The attribute can not be set outside of the constructor.

- The `networkx.generators` module's functions `cycle_graph()`, `path_graph()`, `star_graph()`,

`mesh_graph()`, and `grid_graph()` now have a new kwarg `multigraph` which takes a boolean and defaults to `True`. When it is set to `False` the generated `PyGraph` object will have the `multigraph` attribute set to `False` meaning it will disallow adding parallel edges.

- New *Universal Functions* that can take in a `PyGraph` or `PyDiGraph` instead of being class specific have been to the networkx API. These new functions are:

- `networkx.distance_matrix()`
- `networkx.floyd_warshall_numpy()`
- `networkx.adjacency_matrix()`
- `networkx.all_simple_paths()`
- `networkx.astar_shortest_path()`
- `networkx.dijkstra_shortest_paths()`
- `networkx.dijkstra_shortest_path_lengths()`
- `networkx.k_shortest_path_lengths()`
- `networkx.dfs_edges()`

- Starting with this release wheels will be published for macOS arm64. Only Python 3.9 is supported at first, because it is the only version of Python with native support for arm64 macOS.
- The custom return types `BFSSuccessors`, `NodeIndices`, `EdgeList`, and `WeightedEdgeList` now implement `__str__` so that running `str()` (for example when calling `print()` on the object) it will return a human readable string with the contents of the custom return type.
- The custom return types `BFSSuccessors`, `NodeIndices`, `EdgeList`, and `WeightedEdgeList` now implement `__hash__` so that running `hash()` (for example when insert them into a dict) will return a valid hash for the object. The only exception to this is for `BFSSuccessors` and `WeightedEdgeList` if they contain a Python object that is not hashable, in those cases calling `hash()` will raise a `TypeError`, just like as you called `hash()` on the inner unhashable object.
- Two new methods, `update_edge()` and `update_edge_by_index()` were added to the `networkx.PyDiGraph` and `networkx.PyGraph` (`update_edge()` and `update_edge_by_index()`) classes. These methods are used to update the data payload/weight of an edge in the graph either by the nodes of an edge or by edge index.

### 3.1.3 Bug Fixes

- In previous releases the Python garbage collector did not know how to interact with `PyDiGraph` or `PyGraph` objects and as a result they may never have been freed until Python exited. To fix this issue, the `PyDiGraph` and `PyGraph` classes now are integrated with Python's garbage collector so they'll properly be cleared when there are no more references to a graph object.
- The output from `networkx.PyDiGraph.neighbors()` and `networkx.PyGraph.neighbors()` methods will no longer include duplicate entries in case of parallel edges between nodes. See #250 for more details.
- In previous releases the Python garbage collector did not know how to interact with the custom return types `BFSSuccessors`, `NodeIndices`, `EdgeList`, and `WeightedEdgeList` and as a result they may never have been freed until Python exited. To fix this issue the custom return type classes now are integrated with Python's garbage collector so they'll properly be cleared when there are no more Python references to an object.

## 3.2 0.7.2

### 3.2.1 Bug Fixes

- Fixed a potential segfault that could occur when calling `is_directed_acyclic_graph()` with a a very deep `PyDiGraph` object as reported in [Qiskit/qiskit-terra#5502](#).



## 0.7.1

This release includes a fix for an oversight in the previous 0.7.0 and 0.6.0 releases. Those releases both added custom return types *BFSSuccessors*, *NodeIndices*, *EdgeList*, and *WeightedEdgeList* that implemented the Python sequence protocol which were used in place of lists for certain functions and methods. However, none of those classes had support for being pickled, which was causing compatibility issues for users that were using the return in a context where it would be pickled (for example as an argument to or return of a function called with multiprocessing). This release has a single change over 0.7.0 which is to add the missing support for pickling *BFSSuccessors*, *NodeIndices*, *EdgeList*, and *WeightedEdgeList* which fixes that issue.





This release includes several new features and bug fixes.

This release also dropped support for Python 3.5. If you want to use networkx with Python 3.5 that last version which supports Python 3.5 is 0.6.0.

## 5.1 New Features

- New generator functions for two new generator types, mesh and grid were added to `networkx`. generators for generating all to all and grid graphs respectively. These functions are: `mesh_graph()`, `directed_mesh_graph()`, `grid_graph()`, and `directed_grid_graph()`
- A new function, `networkx.digraph_union()`, for taking the union between two `PyDiGraph` objects has been added.
- A new `PyDiGraph` method `merge_nodes()` has been added. This method can be used to merge 2 nodes in a graph if they have the same weight/data payload.
- A new `PyDiGraph` method `find_node_by_weight()` which can be used to lookup a node index by a given weight/data payload.
- A new return type `NodeIndices` has been added. This class is returned by functions and methods that return a list of node indices. It implements the Python sequence protocol and can be used as list.
- Two new return types `EdgeList` and `WeightedEdgeList`. These classes are returned from functions and methods that return a list of edge tuples and a list of edge tuples with weights. They both implement the Python sequence protocol and can be used as a list
- A new function `collect_runs()` has been added. This function is used to find linear paths of nodes that match a given condition.

## 5.2 Upgrade Notes

- Support for running networkx on Python 3.5 has been dropped. The last release with support for Python 3.5 is 0.6.0.
- The `networkx.PyDiGraph.node_indexes()`, `networkx.PyDiGraph.neighbors()`, `networkx.PyDiGraph.successor_indices()`, `networkx.PyDiGraph.predecessor_indices()`, `networkx.PyDiGraph.add_nodes_from()`, `networkx.PyGraph.node_indexes()`, `networkx.PyGraph.add_nodes_from()`, and `networkx.PyGraph.neighbors()` methods and the `dag_longest_path()`, `topological_sort()`, `graph_astar_shortest_path()`, and `digraph_astar_shortest_path()` functions now return a `NodeIndices` object instead of a list of integers. This should not require any changes unless explicit type checking for a list was used.

- The `networkx.PyDiGraph.edge_list()`, and `networkx.PyGraph.edge_list()` methods and `digraph_dfs_edges()`, `graph_dfs_edges()`, and `digraph_find_cycle()` functions now return an `EdgeList` object instead of a list of integers. This should not require any changes unless explicit type checking for a list was used.
- The `networkx.PyDiGraph.weighted_edge_list()`, `networkx.PyDiGraph.in_edges()`, `networkx.PyDiGraph.out_edges()`, and `networkx.PyGraph.weighted_edge_list` methods now return a `WeightedEdgeList` object instead of a list of integers. This should not require any changes unless explicit type checking for a list was used.

## 5.3 Fixes

- `BFSSuccessors` objects now can be compared with `==` and `!=` to any other Python sequence type.
- The built and published sdist packages for networkx were previously not including the `Cargo.lock` file. This meant that the reproducible build versions of the rust dependencies were not passed through to source. This has been fixed so building from sdist will always use known working versions that we use for testing in CI.

This release includes a number of new features and bug fixes. The main focus of this release was to expand the networkx API functionality to include some commonly needed functions that were missing.

This release is also the first release to provide full support for running with Python 3.9. On previous releases Python 3.9 would likely work, but it would require building networkx from source. Also this will likely be the final release that supports Python 3.5.

## 6.1 New Features

- Two new functions, `digraph_k_shortest_path_lengths()` and `graph_k_shortest_path_lengths()`, for finding the k shortest path lengths from a node in a `PyDiGraph` and `PyGraph`.
- A new method, `is_symmetric()`, to the `PyDiGraph` class. This method will check whether the graph is symmetric or not
- A new kwarg, `as_undirected`, was added to the `digraph_floyd_warshall_numpy()` function. This can be used to treat the input `PyDiGraph` object as if it was undirected for the generated output matrix.
- A new function, `digraph_find_cycle()`, which will return the first cycle during a depth first search of a `PyDiGraph` object.
- Two new functions, `directed_gnm_random_graph()` and `undirected_gnm_random_graph()`, for generating random  $G(n, m)$  graphs.
- A new method, `remove_edges_from()`, was added to `PyDiGraph` and `PyGraph` (`removed_edges_from()`). This can be used to remove multiple edges from a graph object in a single call.
- A new method, `subgraph()`, was added to `PyDiGraph` and `PyGraph` (`subgraph()`) which takes in a list of node indices and will return a new object of the same type representing a subgraph containing the node indices in that list.
- Support for running with Python 3.9
- A new method, `to_undirected()`, was added to `PyDiGraph`. This method will generate an undirected `PyGraph` object from the `PyDiGraph` object.
- A new kwarg, `bidirectional`, was added to the directed generator functions `directed_cycle_graph()`, `directed_path_graph()`, and `directed_star_graph()`. When set to True the directed graphs generated by these functions will add edges in both directions.
- Added two new functions, `is_weakly_connected()` and `weakly_connected_components()`, which will either check if a `PyDiGraph` object is weakly connected or return the list of the weakly connected components of an input `PyDiGraph`.

- The `weight_fn` kwarg for `graph_adjacency_matrix()`, `digraph_adjacency_matrix()`, `graph_floyd_warshall_numpy()`, and `digraph_floyd_warshall_numpy()` is now optional. Previously, it always had to be specified when calling these function. But, instead you can now rely on a default weight float (which defaults to 1.0) to be used for all the edges in the graph.
- Add a `neighbors()` method to `PyGraph` and `PyDiGraph` (`neighbors()`). This function will return the node indices of the neighbor nodes for a given input node.
- Two new methods, `successor_indices()` and `predecessor_indices()`, were added to `PyDiGraph`. These methods will return the node indices for the successor and predecessor nodes of a given input node.
- Two new functions, `graph_distance_matrix()` and `digraph_distance_matrix()`, were added for generating a distance matrix from an input `PyGraph` and `PyDiGraph`.
- Two new functions, `digraph_dijkstra_shortest_paths()` and `graph_dijkstra_shortest_path()`, were added for returning the shortest paths from a node in a `PyDiGraph` and a `PyGraph` object.
- Four new methods, `insert_node_on_in_edges()`, `insert_node_on_out_edges()`, `insert_node_on_in_edges_multiple()`, and `insert_node_on_out_edges_multiple()` were added to `PyDiGraph`. These functions are used to insert an existing node in between an reference node(s) and all it's predecessors or successors.
- Two new functions, `graph_dfs_edges()` and `digraph_dfs_edges()`, were added to get an edge list in depth first order from a `PyGraph` and `PyDiGraph`.

## 6.2 Upgrade Notes

- The numpy arrays returned by `graph_floyd_warshall_numpy()`, `digraph_floyd_warshall_numpy()`, `digraph_adjacency_matrix()`, and `graph_adjacency_matrix()` will now be in a contiguous C array memory layout. Previously, they would return arrays in a column-major fortran layout. This was change was made to make it easier to interface the arrays returned by these functions with other C Python extensions. There should be no change when interacting with the numpy arrays via numpy's API.
- The `bfs_successors()` method now returns an object of a custom type `BFSSuccessors` instead of a list. The `BFSSuccessors` type implements the Python sequence protocol so it can be used in place like a list (except for where explicit type checking is used). This was done to defer the type conversion between Rust and Python since doing it all at once can be a performance bottleneck especially for large graphs. The `BFSSuccessors` class will only do the type conversion when an element is accessed.

## 6.3 Fixes

- When pickling `PyDiGraph` objects the original node indices will be preserved across the pickle.
- The random  $G(n, p)$  functions, `directed_gnp_random_graph()` and `undirected_gnp_random_graph()`, will now also handle exact 0 or 1 probabilities. Previously it would fail in these cases. Fixes #172

This release include a number of new features and bug fixes. The main focus of the improvements of this release was to increase the ease of interacting with graph objects. This includes adding support for generating dot output which can be used with graphviz (or similar tools) for visualizing graphs adding more methods to query the state of graph, adding a generator module for easily creating graphs of certain shape, and implementing the mapping protocol so you can directly interact with graph objects.

## 7.1 New Features

- A new method, `to_dot()`, was added to `PyGraph` and `PyDiGraph` (`to_dot()`). It will generate a dot format representation of the object which can be used with Graphviz (or similar tooling) to generate visualizations of graphs.
- Added a new function, `strongly_connected_components()`, to get the list of strongly connected components of a `PyDiGraph` object.
- A new method, `compose()`, for composing another graph object of the same type into a graph was added to `PyGraph` and `PyDiGraph` (`compose()`).
- The `PyGraph` and `PyDiGraph` classes now implement the Python mapping protocol for interacting with graph nodes. You can now access and interact with node data directly by using standard map access patterns in Python. For example, accessing a graph like `graph[1]` will return the weight/data payload for the node at index 1.
- A new module, `networkx.generators`, has been added. Functions in this module can be used for quickly generating graphs of certain shape. To start it includes:
  - `networkx.generators.cycle_graph()`
  - `networkx.generators.directed_cycle_graph()`
  - `networkx.generators.path_graph()`
  - `networkx.generators.directed_path_graph()`
  - `networkx.generators.star_graph()`
  - `networkx.generators.directed_star_graph()`
- A new method, `remove_node_retain_edges()`, has been added to the `PyDiGraph` class. This method can be used to remove a node and add edges from its predecessors to its successors.
- Two new methods, `edge_list()` and `weighted_edge_list()`, for getting a list of tuples with the edge source and target (with or without edge weights) have been added to `PyGraph` and `PyDiGraph` (`edge_list()` and `weighted_edge_list()`).
- A new function, `cycle_basis()`, for getting a list of cycles which form a basis for cycles of a `PyGraph` object.

- Two new functions, `graph_floyd_warshall_numpy()` and `digraph_floyd_warshall_numpy()`, were added for running the Floyd Warshall algorithm and returning all the shortest path lengths as a distance matrix.
- A new constructor method, `read_edge_list()`, has been added to `PyGraph` and `PyDigraph` (`read_edge_list()`). This method will take in a path to an edge list file and will read that file and generate a new object from the contents.
- Two new methods, `extend_from_edge_list()` and `extend_from_weighted_edge_list()` has been added to `PyGraph` and `PyDiGraph` (`extend_from_edge_list()` and `extend_from_weighted_edge_list()`). This method takes in an edge list and will add both the edges and nodes (if a node index used doesn't exist yet) in the list to the graph.

## 7.2 Fixes

- The limitation with the `is_isomorphic()` and `is_isomorphic_node_match()` functions that would cause segfaults when comparing graphs with node removals has been fixed. You can now run either function with any `PyDiGraph/PyDAG` objects, even if there are node removals. Fixes #27
- If an invalid node index was passed as part of the `first_layer` argument to the `layers()` function it would previously raise a `PanicException` that included a Rust backtrace and no other user actionable details which was caused by an unhandled error. This has been fixed so that an `IndexError` is raised and the problematic node index is included in the exception message.

This release includes many new features and fixes, including improved performance and better documentation. But, the biggest change for this release is that this is the first release of `networkx` that supports compilation with a stable released version of rust. This was made possible thanks to all the hard work of the PyO3 maintainers and contributors in the PyO3 0.11.0 release.

## 8.1 New Features

- A new class for undirected graphs, `PyGraph`, was added.
- 2 new functions `graph_adjacency_matrix()` and `digraph_adjacency_matrix()` to get the adjacency matrix of a `PyGraph` and `PyDiGraph` object.
- A new `PyDiGraph` method, `find_adjacent_node_by_edge()`, was added. This is used to locate an adjacent node given a condition based on the edge between them.
- New methods, `add_nodes_from()`, `add_edges_from()`, `add_edges_from_no_data()`, and `remove_nodes_from()` were added to `PyDiGraph`. These methods allow for the addition (and removal) of multiple nodes or edges from a graph in a single call.
- A new function, `graph_greedy_color()`, which is used to return a coloring map from a `PyGraph` object.
- 2 new functions, `graph_astar_shortest_path()` and `digraph_astar_shortest_path()`, to find the shortest path from a node to a specified goal using the A\* search algorithm.
- 2 new functions, `graph_all_simple_paths()` and `digraph_all_simple_paths()`, to return a list of all the simple paths between 2 nodes in a `PyGraph` or a `PyDiGraph` object.
- 2 new functions, `directed_gnp_random_graph()` and `undirected_gnp_random_graph()`, to generate  $G_{np}$  random `PyDiGraph` and `PyGraph` objects.
- 2 new functions, `graph_dijkstra_shortest_path_lengths()` and `digraph_dijkstra_shortest_path_lengths()`, were added for find the shortest path length between nodes in `PyGraph` or `PyDiGraph` object using Dijkstra's algorithm.

## 8.2 Upgrade Notes

- The *PyDAG* class was renamed *PyDiGraph* to better reflect its functionality. For backwards compatibility *PyDAG* still exists as a Python subclass of *PyDiGraph*. No changes should be required for existing users.
- *numpy* is now a dependency of *networkx*. This is used for the adjacency matrix functions to return numpy arrays. The minimum version of *numpy* supported is 1.16.0.

## 8.3 Fixes

- The *networkx* exception classes are now properly exported from the *networkx* module. In prior releases it was not possible to import the exception classes (normally to catch one being raised) requiring users to catch the base *Exception* class. This has been fixed so a specialized *networkx* exception class can be used.



## CONTRIBUTING

First read the overall Qiskit project contribution guidelines. These are all included in the Qiskit documentation:

[https://qiskit.org/documentation/contributing\\_to\\_qiskit.html](https://qiskit.org/documentation/contributing_to_qiskit.html)

While it's not all directly applicable since most of it is about the Qiskit project itself and `retworkx` is an independent library developed in tandem with Qiskit; the general guidelines and advice still apply here.

### 9.1 Contributing to `retworkx`

In addition to the general guidelines there are specific details for contributing to `retworkx`, these are documented below.

#### 9.1.1 Tests

Once you've made a code change, it is important to verify that your change does not break any existing tests and that any new tests that you've added also run successfully. Before you open a new pull request for your change, you'll want to run the test suite locally.

The easiest way to run the test suite is to use `**tox**`. You can install `tox` with `pip`: `pip install -U tox`. `Tox` provides several advantages, but the biggest one is that it builds an isolated virtualenv for running tests. This means it does not pollute your system python when running.

Note, if you run tests outside of `tox` that you can **not** run the tests from the root of the repo, this is because `retworkx` packaging shim will conflict with imports from `retworkx` the installed version of `retworkx` (which contains the compiled extension).

#### 9.1.2 Style

##### Rust

Rust is the primary language of `retworkx` and all the functional code in the libraries is written in Rust. The Rust code in `retworkx` uses `rustfmt` to enforce consistent style. CI jobs are configured to ensure to check this. Luckily adapting your code is as simple as running:

```
cargo fmt
```

locally. This will automatically restyle the rust code in `retworkx` to match what CI is checking.

## Lint

An additional step is to run `clippy` on your changes. While this is not run in CI (because it's very dependent on the rust/cargo version) it can often catch issues in your code. You can run it by running:

```
cargo clippy
```

## Python

Python is used primarily for tests and some small pieces of packaging and namespace configuration code in the actual library. `flake8` is used to enforce consistent style in the python code in the repository. You can run it via tox using:

```
tox -elint
```

This will also run `cargo fmt` in check mode to ensure that you ran `cargo fmt` and will fail if the Rust code doesn't conform to the style rules.

### 9.1.3 Building documentation

Just like with tests building documentation is done via tox. This will handle compiling retworkx, installing the python dependencies, and then building the documentation in an isolated venv. You can run just the docs build with:

```
tox -edocs
```

which will output the html rendered documentation in `docs/build/html` which you can view locally in a web browser.

### 9.1.4 Release Notes

It is important to document any end user facing changes when we release a new version of retworkx. The expectation is that if your code contribution has user facing changes that you will write the release documentation for these changes. This documentation must explain what was changed, why it was changed, and how users can either use or adapt to the change. The idea behind release documentation is that when a naive user with limited internal knowledge of the project is upgrading from the previous release to the new one, they should be able to read the release notes, understand if they need to update their program which uses retworkx, and how they would go about doing that. It ideally should explain why they need to make this change too, to provide the necessary context.

To make sure we don't forget a release note or if the details of user facing changes over a release cycle we require that all user facing changes include documentation at the same time as the code. To accomplish this we use the `reno` tool which enables a git based workflow for writing and compiling release notes.

#### Adding a new release note

Making a new release note is quite straightforward. Ensure that you have `reno` installed with:

```
pip install -U reno
```

Once you have `reno` installed you can make a new release note by running in your local repository checkout's root:

```
reno new short-description-string
```

where short-description-string is a brief string (with no spaces) that describes what's in the release note. This will become the prefix for the release note file. Once that is run it will create a new yml file in releasenotes/notes. Then open that yml file in a text editor and write the release note. The basic structure of a release note is restructured text in yml lists under category keys. You add individual items under each category and they will be grouped automatically by release when the release notes are compiled. A single file can have as many entries in it as needed, but to avoid potential conflicts you'll want to create a new file for each pull request that has user facing changes. When you open the newly created file it will be a full template of the different categories with a description of a category as a single entry in each category. You'll want to delete all the sections you aren't using and update the contents for those you are. For example, the end result should look something like:

#### features:

- |  
Added a new function, `:func:`~networkx.foo`` that adds support for doing something to `:class:`~networkx.PyDiGraph`` objects.
- |  
The `:class:`~networkx.PyDiGraph`` class has a new method `:meth:`~networkx.PyDiGraph.foo``. This is the equivalent of calling the `:func:`~networkx.foo`` function to do something to your `:class:`~networkx.PyDiGraph`` object, but provides the convenience of running it natively on an object. For example::  
  

```
from networkx import PyDiGraph

g = PyDiGraph()
g.foo()
```

#### deprecations:

- |  
The ```networkx.bar``` function has been deprecated and will be removed in a future release. It has been superseded by the `:meth:`~networkx.PyDiGraph.foo`` method and `:func:`~networkx.foo`` function which provides similar functionality but with more accurate results and better performance. You should update your calls ```networkx.bar()``` calls to use ```networkx.foo()``` instead.

You can also look at other release notes for other examples.

You can use any [sphinx feature](#) in them (code sections, tables, enumerated lists, bulleted list, etc) to express what is being changed as needed. In general you want the release notes to include as much detail as needed so that users will understand what has changed, why it changed, and how they'll have to update their code.

After you've finished writing your release notes you'll want to add the note file to your commit with `git add` and commit them to your PR branch to make sure they're included with the code in your PR.

## Linking to issues

If you need to link to an issue or other Github artifact as part of the release note this should be done using an inline link with the text being the issue number. For example you would write a release note with a link to issue 12345 as:

#### fixes:

- |  
Fixes a race condition in the function ```foo()```. Refer to [#12345 <https://github.com/Qiskit/networkx/issues/12345>](https://github.com/Qiskit/networkx/issues/12345) for more details.

### Generating the release notes

After release notes have been added if you want to see what the full output of the release notes. Reno is used to combine the release note yaml files into a single rst (ReStructuredText) document that `sphinx` will then compile for us as part of the documentation builds. If you want to generate the rst file you use the `reno report` command. If you want to generate the full retworkx release notes for all releases (since we started using reno during 0.8) you just run:

```
reno report
```

but you can also use the `--version` argument to view a single release (after it has been tagged):

```
reno report --version 0.8.0
```

### Building release notes locally

Building the release notes is part of the standard retworkx documentation builds. To check what the rendered html output of the release notes will look like for the current state of the repo you can run: `tox -edocs` which will build all the documentation into `docs/_build/html` and the release notes in particular will be located at `docs/_build/html/release_notes.html`

**Warning:** The documentation for retworkx has migrated to:

<https://qiskit.org/documentation/retworkx>

These docs will no longer be updated.

## NETWORKX FOR NETWORKX USERS

This is an introductory guide for existing networkx users on how to use networkx, how it differs from networkx, and key differences to keep in mind.

### 10.1 Some Key Differences

networkx (as the name implies) was inspired by networkx and the goal of the project is to provide a similar level of functionality and utility to what networkx offers but with much faster performance. However, because of limitations in the boundary between rust and python, different design decisions, and other differences the libraries are not identical.

The biggest difference to keep in mind is networkx is a very dynamic in how it can be used. It allows you to treat a graph object associatively (like a python dictionary) and interact with the graph using the objects you're putting on the graph. For example:

```
import networkx as nx

graph = nx.MultiDiGraph()
graph.add_node('my_node_a')
graph.add_node('my_node_b')
graph.add_edge('my_node_a', 'my_node_b')
```

While networkx being written in Rust puts more constraints on how you interact with graph objects. With networkx you can still attach any Python object on the a graph but each node and edge is assigned an integer index. That index must be used for accessing nodes and edges on the graph. In networkx the above example would be something like:

```
import networkx as rx

graph = rx.PyDiGraph()
node_a = graph.add_node('my_node_a')
node_b = graph.add_node('my_node_b')
graph.add_edge(node_a, node_b, None)
```

where `node_a == 0` and `node_b == 1`. These node indices can be used with a graph object to access the objects set as the payload object via the python mapping protocol (**not** the sequence protocol because the indices are not guaranteed to be a sequence after nodes or edges are removed from a graph). Continuing from the above networkx example:

```
assert 'my_node_a' == graph[node_a]
assert 'my_node_b' == graph[node_b]
```

The use of integer indexes for everything is normally the biggest difference that existing networkx users have to adapt to when migrating to networkx.

Similarly when there are algorithm functions that operate on a node or edge data, callback functions are used in networkx to return statically typed data from node or edge payloads to use for various algorithms. In networkx this is typically done using named attributes of nodes or edges (the typical example of a node or edge attribute named `weight` is used by default for functions that need a numerical weight).

For example, in networkx:

```
import networkx as nx

graph = nx.MultiDiGraph()
graph.add_edges_from([(0, 1, {'weight': 1}), (0, 2, {'weight': 2}),
                    (1, 3, {'weight': 2}), (3, 0, {'weight': 3})])
dist_matrix = nx.floyd_warshall_numpy(graph, weight='weight')
```

while in networkx you would use:

```
import networkx as rx

graph = rx.PyDiGraph()
graph.extend_from_weighted_edge_list(
    [(0, 1, {'weight': 1}), (0, 2, {'weight': 2}),
     (1, 3, {'weight': 2}), (3, 0, {'weight': 3})])
dist_matrix = rx.digraph_floyd_warshall_numpy(
    graph, weight_fn=lambda edge: edge[weight])
```

or more concisely:

```
import networkx as rx

graph = rx.PyDiGraph()
graph.extend_from_weighted_edge_list(
    [(0, 1, 1), (0, 2, 2),
     (1, 3, 2), (3, 0, 3)])
dist_matrix = rx.digraph_floyd_warshall_numpy(graph,
                                             weight_fn=lambda edge: edge)
```

The other large difference to keep in mind is that most functions in networkx are explicitly typed. This means that they either always return or accept either a `PyDiGraph` or a `PyGraph` but not both. The exception to this are the *Universal Functions* which will dispatch to the statically typed equivalent based on the object they receive. This is different from networkx where everything is pretty much dynamically typed and you can pass a graph object to any function and it will work as expected (unless it isn't supported and then it will raise an exception).

## 10.2 Graph Data and Attributes

### 10.2.1 Nodes

In networkx a node can be any hashable python object. That object is then used to access or refer to a node. Additionally, you can set optional attributes on a node. This is described in more detail below.

In networkx any python object (hashable or not) can be used as a node, however nodes can only be accessed by an integer node index (which is returned by any function adding a node). There are no optional attributes for nodes. If this is required that is expected to be added to the node's data payload.

## 10.2.2 Edges

Edges in networkx are accessible by the tuple of the nodes the edge is between. Edges only have optional attributes (as described below) and no other object payload.

In networkx any python object can be an edge and have a unique integer index assigned to it, just like nodes. However, edges are in most functions/methods referenced by the tuple of the indices of the nodes the edge is between instead of the edge's index.

## 10.2.3 Attributes

networkx has a concept of [graph](#), [node](#), and [edge attributes](#) in addition to the hashable object used for a node's payload. Networkx has no analogous concept. Instead, the payloads for nodes and edges are any python object (hashable or not). This enables you to build similar structures to the attributes concept, but also use alternative structures specific to your use case.

For example, something like:

```
import networkx as nx

graph = nx.Graph()
graph.add_node(1, time='5pm')
graph.add_nodes_from([3], time='2pm')
graph.nodes[1]['room'] = 714
```

can be accomplished by using a dict for node weights:

```
import networkx as rx

graph = rx.PyGraph()
node_a = graph.add_node({'time': '5pm'})
node_b = graph.add_nodes_from([{'time': '2pm'}])
graph[node_a]['room'] = 714
```

## 10.2.4 Examining elements of a graph

networkx provides 4 attributes on graph objects `nodes`, `edges`, `adj`, and `degree` which act as set like views for the nodes, edges, neighbors, and degrees of nodes respectively. These properties provide a real time view into the different properties of the graphs and provide additional methods on those attributes for looking at graph properties in different ways.

networkx doesn't offer views, but instead provides different accessor methods that return copies of the analogous data. There are different functions/methods that offer different views on that data. For example, `edge_list()` is analogous to networkx's `edges` view and `weighted_edge_list()` is equivalent to networkx's `edges(data=True)`.

Additionally, since everything in networkx is integer indexed, to access node data the `PyDiGraph` and `PyGraph` classes implement the python mapping protocol so you can access node's data using a node's index.

## 10.3 API Equivalents

### 10.3.1 Class Constructors

networkx	networkx	Notes
Graph()	<code>PyGraph(multigraph=False)</code>	Only in multigraph flag added in networkx >= 0.8.0 prior releases always allow multiple edges
DiGraph()	<code>PyDiGraph(multigraph=False)</code>	Only in multigraph flag added in networkx >= 0.8.0 prior releases always allow multiple edges
MultiGraph()	<code>PyGraph()</code>	
MultiDiGraph()	<code>PyDiGraph()</code>	

The other thing to note here is that networkx does not allow initialization of a graph when the constructor is called. You will need to call an appropriate method of the object to add nodes or edges or use an alternative constructor method:

networkx	networkx	Notes
<code>Graph([(0, 1), (1, 0)])</code>	<code>graph = PyGraph() graph.extend_from_edge_ →list([(0, 1), (1, 0)])</code>	networkx input must be a list of 2-tuples, while networkx can be an iterator
<code>Graph([(0, 1, {'weight': 2} →), (1, 0, {'weight': 1} →)])</code>	<code>graph = PyGraph() graph.extend_from_edge_ →list([(0, 1, 2), (1, 0, →1)])</code>	networkx input must be a list of 3-tuples, while networkx can be an iterator
<code>Graph(np.array([(0, 1, 1), →[1, 0, 1], [1, 0, 1]]))</code>	<code>PyGraph.from_adjacency_ →matrix(np.array([(0, 1, →1], [1, 0, 1], [1, 0, →1]), dtype=np.float64))</code>	networkx <code>from_adjacency_matrix()</code> can only take a float dtype numpy array, you can use <code>.astype(np.float64, copy=False)</code> to adapt a non-float array.

### 10.3.2 Graph Modifiers

net- workx	networkx	Notes
<code>add_node()</code>	<code>add_node()</code>	networkx returns a node index for the newly created node
<code>add_nodes_from()</code>	<code>add_nodes_from()</code>	networkx requires the input to be a list of objects and will return a list of node indices for the newly created nodes
<code>add_edge()</code>	<code>add_edge()</code>	networkx requires 3 parameters be used, the 2 node indices and the payload (networkx works with either 2 or 3)
<code>add_edges_from()</code> , <code>add_edges_from_no_data()</code> , <code>extend_from_edge_list()</code> , <code>extend_from_weighted_edge_list()</code>	<code>add_edges_from()</code> , <code>add_edges_from_no_data()</code> , <code>extend_from_edge_list()</code> , <code>extend_from_weighted_edge_list()</code>	networkx requires a list of either a 3 or 2 tuple (depending on whether weights/data are expected or not). The difference between the networkx <code>extend_from*</code> and <code>add_edges_from*</code> methods are that the <code>extend_from*</code> will create new nodes with a weight/data payload of None if any node indices are missing.

(note the networkx version links to the `PyDiGraph` version, but there are also equivalent `PyGraph` methods available)



## 10.4 Functionality Gaps

networkx is a mature library that has a wide user base and extensive feature set, while retworkx, by comparison, is a much younger library and is missing a lot of the features that networkx offers. If you encounter a feature that networkx offers which is missing from retworkx that you would like to use please open an “Enhancement request” issue at: <https://github.com/Qiskit/retworkx/issues/new/choose> Once an issue is opened we can prioritize working on adding an equivalent feature to retworkx.



## Symbols

\_\_init\_\_() (*BFSSuccessors method*), 105  
 \_\_init\_\_() (*EdgeList method*), 107  
 \_\_init\_\_() (*NodeIndices method*), 106  
 \_\_init\_\_() (*PyDAG method*), 38  
 \_\_init\_\_() (*PyDiGraph method*), 21  
 \_\_init\_\_() (*PyGraph method*), 8  
 \_\_init\_\_() (*WeightedEdgeList method*), 108

## A

add\_child() (*PyDAG method*), 40  
 add\_child() (*PyDiGraph method*), 22  
 add\_edge() (*PyDAG method*), 40  
 add\_edge() (*PyDiGraph method*), 22  
 add\_edge() (*PyGraph method*), 9  
 add\_edges\_from() (*PyDAG method*), 40  
 add\_edges\_from() (*PyDiGraph method*), 23  
 add\_edges\_from() (*PyGraph method*), 9  
 add\_edges\_from\_no\_data() (*PyDAG method*), 40  
 add\_edges\_from\_no\_data() (*PyDiGraph method*), 23  
 add\_edges\_from\_no\_data() (*PyGraph method*), 9  
 add\_node() (*PyDAG method*), 41  
 add\_node() (*PyDiGraph method*), 23  
 add\_node() (*PyGraph method*), 10  
 add\_nodes\_from() (*PyDAG method*), 41  
 add\_nodes\_from() (*PyDiGraph method*), 23  
 add\_nodes\_from() (*PyGraph method*), 10  
 add\_parent() (*PyDAG method*), 41  
 add\_parent() (*PyDiGraph method*), 23  
 adj() (*PyDAG method*), 41  
 adj() (*PyDiGraph method*), 24  
 adj() (*PyGraph method*), 10  
 adj\_direction() (*PyDAG method*), 41  
 adj\_direction() (*PyDiGraph method*), 24  
 adjacency\_matrix() (*in module networkx*), 98  
 all\_simple\_paths() (*in module networkx*), 99  
 ancestors() (*in module networkx*), 79  
 astar\_shortest\_path() (*in module networkx*), 99

## B

bfs\_successors() (*in module networkx*), 75  
 BFSSuccessors (*class in networkx*), 105

## C

check\_cycle (*PyDAG attribute*), 42  
 check\_cycle (*PyDiGraph attribute*), 24  
 collect\_runs() (*in module networkx*), 83  
 compose() (*PyDAG method*), 42  
 compose() (*PyDiGraph method*), 24  
 compose() (*PyGraph method*), 10  
 cycle\_basis() (*in module networkx*), 91  
 cycle\_graph() (*in module networkx.generators*), 55

## D

dag\_longest\_path() (*in module networkx*), 75  
 dag\_longest\_path\_length() (*in module networkx*), 76  
 DAGHasCycle, 104  
 DAGWouldCycle, 103  
 degree() (*PyGraph method*), 13  
 descendants() (*in module networkx*), 79  
 dfs\_edges() (*in module networkx*), 102  
 digraph\_adjacency\_matrix() (*in module networkx*), 84  
 digraph\_all\_simple\_paths() (*in module networkx*), 86  
 digraph\_astar\_shortest\_path() (*in module networkx*), 87  
 digraph\_dfs\_edges() (*in module networkx*), 93  
 digraph\_dijkstra\_shortest\_path\_lengths() (*in module networkx*), 89  
 digraph\_dijkstra\_shortest\_paths() (*in module networkx*), 88  
 digraph\_distance\_matrix() (*in module networkx*), 81  
 digraph\_find\_cycle() (*in module networkx*), 93  
 digraph\_floyd\_warshall\_numpy() (*in module networkx*), 82  
 digraph\_k\_shortest\_path\_lengths() (*in module networkx*), 90  
 digraph\_union() (*in module networkx*), 94  
 dijkstra\_shortest\_path\_lengths() (*in module networkx*), 101  
 dijkstra\_shortest\_paths() (*in module networkx*), 100  
 directed\_cycle\_graph() (*in module networkx.generators*), 56

directed\_gnm\_random\_graph() (in module networkx), 72  
 directed\_gnp\_random\_graph() (in module networkx), 71  
 directed\_grid\_graph() (in module networkx.generators), 69  
 directed\_mesh\_graph() (in module networkx.generators), 66  
 directed\_path\_graph() (in module networkx.generators), 60  
 directed\_star\_graph() (in module networkx.generators), 63  
 distance\_matrix() (in module networkx), 96

## E

edge\_list() (PyDAG method), 44  
 edge\_list() (PyDiGraph method), 27  
 edge\_list() (PyGraph method), 13  
 EdgeList (class in networkx), 107  
 edges() (PyDAG method), 44  
 edges() (PyDiGraph method), 27  
 edges() (PyGraph method), 13  
 extend\_from\_edge\_list() (PyDAG method), 44  
 extend\_from\_edge\_list() (PyDiGraph method), 27  
 extend\_from\_edge\_list() (PyGraph method), 13  
 extend\_from\_weighted\_edge\_list() (PyDAG method), 45  
 extend\_from\_weighted\_edge\_list() (PyDiGraph method), 27  
 extend\_from\_weighted\_edge\_list() (PyGraph method), 13

## F

find\_adjacent\_node\_by\_edge() (PyDAG method), 45  
 find\_adjacent\_node\_by\_edge() (PyDiGraph method), 28  
 find\_node\_by\_weight() (PyDAG method), 45  
 find\_node\_by\_weight() (PyDiGraph method), 28  
 floyd\_warshall() (in module networkx), 81  
 floyd\_warshall\_numpy() (in module networkx), 97  
 from\_adjacency\_matrix() (PyDAG static method), 45  
 from\_adjacency\_matrix() (PyDiGraph static method), 28  
 from\_adjacency\_matrix() (PyGraph static method), 14

## G

get\_all\_edge\_data() (PyDAG method), 45  
 get\_all\_edge\_data() (PyDiGraph method), 28  
 get\_all\_edge\_data() (PyGraph method), 14  
 get\_edge\_data() (PyDAG method), 46  
 get\_edge\_data() (PyDiGraph method), 28  
 get\_edge\_data() (PyGraph method), 14

get\_node\_data() (PyDAG method), 46  
 get\_node\_data() (PyDiGraph method), 29  
 get\_node\_data() (PyGraph method), 14  
 graph\_adjacency\_matrix() (in module networkx), 85  
 graph\_all\_simple\_paths() (in module networkx), 85  
 graph\_astar\_shortest\_path() (in module networkx), 86  
 graph\_dfs\_edges() (in module networkx), 92  
 graph\_dijkstra\_shortest\_path\_lengths() (in module networkx), 89  
 graph\_dijkstra\_shortest\_paths() (in module networkx), 88  
 graph\_distance\_matrix() (in module networkx), 80  
 graph\_floyd\_warshall\_numpy() (in module networkx), 82  
 graph\_greedy\_color() (in module networkx), 91  
 graph\_k\_shortest\_path\_lengths() (in module networkx), 90  
 grid\_graph() (in module networkx.generators), 68

## H

has\_edge() (PyDAG method), 46  
 has\_edge() (PyDiGraph method), 29  
 has\_edge() (PyGraph method), 15

## I

in\_degree() (PyDAG method), 46  
 in\_degree() (PyDiGraph method), 29  
 in\_edges() (PyDAG method), 46  
 in\_edges() (PyDiGraph method), 29  
 insert\_node\_on\_in\_edges() (PyDAG method), 46  
 insert\_node\_on\_in\_edges() (PyDiGraph method), 29  
 insert\_node\_on\_in\_edges\_multiple() (PyDAG method), 47  
 insert\_node\_on\_in\_edges\_multiple() (PyDiGraph method), 30  
 insert\_node\_on\_out\_edges() (PyDAG method), 47  
 insert\_node\_on\_out\_edges() (PyDiGraph method), 30  
 insert\_node\_on\_out\_edges\_multiple() (PyDAG method), 47  
 insert\_node\_on\_out\_edges\_multiple() (PyDiGraph method), 30  
 InvalidNode, 103  
 is\_directed\_acyclic\_graph() (in module networkx), 77  
 is\_isomorphic() (in module networkx), 77  
 is\_isomorphic\_node\_match() (in module networkx), 78  
 is\_matching() (in module networkx), 94  
 is\_maximal\_matching() (in module networkx), 95  
 is\_symmetric() (PyDAG method), 47  
 is\_symmetric() (PyDiGraph method), 30

`is_weakly_connected()` (in module *networkx*), 77

## K

`k_shortest_path_lengths()` (in module *networkx*), 101

## L

`layers()` (in module *networkx*), 84

`lexicographical_topological_sort()` (in module *networkx*), 80

## M

`max_weight_matching()` (in module *networkx*), 95

`merge_nodes()` (*PyDAG* method), 47

`merge_nodes()` (*PyDiGraph* method), 30

`mesh_graph()` (in module *networkx.generators*), 65

`multigraph` (*PyDAG* attribute), 48

`multigraph` (*PyDiGraph* attribute), 30

`multigraph` (*PyGraph* attribute), 15

## N

`neighbors()` (*PyDAG* method), 48

`neighbors()` (*PyDiGraph* method), 31

`neighbors()` (*PyGraph* method), 15

`node_indexes()` (*PyDAG* method), 48

`node_indexes()` (*PyDiGraph* method), 31

`node_indexes()` (*PyGraph* method), 15

`NodeIndices` (class in *networkx*), 106

`nodes()` (*PyDAG* method), 48

`nodes()` (*PyDiGraph* method), 31

`nodes()` (*PyGraph* method), 15

`NoEdgeBetweenNodes`, 103

`NoPathFound`, 104

`NoSuitableNeighbors`, 104

`NullGraph`, 105

`number_weakly_connected_components()` (in module *networkx*), 76

## O

`out_degree()` (*PyDAG* method), 48

`out_degree()` (*PyDiGraph* method), 31

`out_edges()` (*PyDAG* method), 48

`out_edges()` (*PyDiGraph* method), 31

## P

`path_graph()` (in module *networkx.generators*), 58

`predecessor_indices()` (*PyDAG* method), 48

`predecessor_indices()` (*PyDiGraph* method), 31

`predecessors()` (*PyDAG* method), 49

`predecessors()` (*PyDiGraph* method), 31

*PyDAG* (class in *networkx*), 37

*PyDiGraph* (class in *networkx*), 20

*PyGraph* (class in *networkx*), 7

## R

`read_edge_list()` (*PyDAG* static method), 49

`read_edge_list()` (*PyDiGraph* static method), 32

`read_edge_list()` (*PyGraph* static method), 15

`remove_edge()` (*PyDAG* method), 50

`remove_edge()` (*PyDiGraph* method), 33

`remove_edge()` (*PyGraph* method), 16

`remove_edge_from_index()` (*PyDAG* method), 50

`remove_edge_from_index()` (*PyDiGraph* method), 33

`remove_edge_from_index()` (*PyGraph* method), 17

`remove_edges_from()` (*PyDAG* method), 50

`remove_edges_from()` (*PyDiGraph* method), 33

`remove_edges_from()` (*PyGraph* method), 17

`remove_node()` (*PyDAG* method), 50

`remove_node()` (*PyDiGraph* method), 33

`remove_node()` (*PyGraph* method), 17

`remove_node_retain_edges()` (*PyDAG* method), 50

`remove_node_retain_edges()` (*PyDiGraph* method), 33

`remove_nodes_from()` (*PyDAG* method), 51

`remove_nodes_from()` (*PyDiGraph* method), 34

`remove_nodes_from()` (*PyGraph* method), 17

## S

`star_graph()` (in module *networkx.generators*), 62

`strongly_connected_components()` (in module *networkx*), 92

`subgraph()` (*PyDAG* method), 51

`subgraph()` (*PyDiGraph* method), 34

`subgraph()` (*PyGraph* method), 17

`successor_indices()` (*PyDAG* method), 51

`successor_indices()` (*PyDiGraph* method), 34

`successors()` (*PyDAG* method), 51

`successors()` (*PyDiGraph* method), 34

## T

`to_dot()` (*PyDAG* method), 51

`to_dot()` (*PyDiGraph* method), 34

`to_dot()` (*PyGraph* method), 17

`to_undirected()` (*PyDAG* method), 53

`to_undirected()` (*PyDiGraph* method), 36

`topological_sort()` (in module *networkx*), 78

## U

`undirected_gnm_random_graph()` (in module *networkx*), 73

`undirected_gnp_random_graph()` (in module *networkx*), 72

`update_edge()` (*PyDAG* method), 53

`update_edge()` (*PyDiGraph* method), 36

`update_edge()` (*PyGraph* method), 19

`update_edge_by_index()` (*PyDAG* method), 54

`update_edge_by_index()` (*PyDiGraph* method), 36

`update_edge_by_index()` (*PyGraph method*), 19

## W

`weakly_connected_components()` (*in module networkx*), 76

`weighted_edge_list()` (*PyDAG method*), 54

`weighted_edge_list()` (*PyDiGraph method*), 37

`weighted_edge_list()` (*PyGraph method*), 19

`WeightedEdgeList` (*class in networkx*), 107

`with_traceback()` (*DAGHasCycle method*), 104

`with_traceback()` (*DAGWouldCycle method*), 103

`with_traceback()` (*InvalidNode method*), 103

`with_traceback()` (*NoEdgeBetweenNodes method*), 103

`with_traceback()` (*NoPathFound method*), 104

`with_traceback()` (*NoSuitableNeighbors method*), 104

`with_traceback()` (*NullGraph method*), 105