retworkx Documentation

Release 0.8.0

Matthew Treinish

May 24, 2021

CONTENTS

Contents:

CHAPTER

ONE

RETWORKX

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

retworkx 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 retworkx

retworkx 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 retworkx
```

This will install a precompiled version of retworkx 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 able to build the package from the published source package you need to have rust >=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 retworkx
```

will build retworkx 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 then 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"])
```

CHAPTER

TWO

RETWORKX API REFERENCE

2.1 Graph Classes

retworkx.PyGraph([multigraph])	A class for creating undirected graphs
retworkx.PyDiGraph([check_cycle, multigraph])	A class for creating directed graphs
retworkx.PyDAG([check_cycle, multigraph])	A class for creating direct acyclic graphs.

2.1.1 retworkx.PyGraph

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

A class for creating undirected graphs

The PyGraph 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 retworkx

graph = retworkx.PyGraph()
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 retworkx

graph = retworkx.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

__init__()	Initialize self.
add_edge(node_a, node_b, edge, /)	Add an edge between 2 nodes.
add_edges_from(obj_list, /)	Add new edges to the graph.
add_edges_from_no_data(obj_list, /)	Add new edges to the graph without python data.
add_node(obj, /)	Add a new node to the graph.
add_nodes_from(obj_list, /)	Add new nodes to the graph.
adj(node, /)	Get the index and data for the neighbors of a node.
compose(other, node_map, /[, node_map_func, ...])	Add another PyGraph object into this PyGraph
degree(node, /)	Get the degree for a node
edge_list()	Get edge list
edges()	Return a list of all edge data.
extend_from_edge_list(edge_list, /)	Extend graph from an edge list
extend_from_weighted_edge_list(edge_lsit, /)	Extend graph from a weighted edge list
from_adjacency_matrix(matrix, /)	Create a new *PyGraph* object from an adjacency matrix
get_all_edge_data(node_a, node_b, /)	Return the edge data for all the edges between 2 nodes.
get_edge_data(node_a, node_b, /)	Return the edge data for the edge between 2 nodes.
get_node_data(node, /)	Return the node data for a given node index
has_edge(node_a, node_b, /)	Return True if there is an edge between node_a to node_b.
neighbors(node, /)	Get the neighbors of a node.
node_indexes()	Return a list of all node indexes.
nodes()	Return a list of all node data.
read_edge_list(path, /[, comment, deliminator])	Read an edge list file and create a new PyGraph object from the contents
remove_edge(node_a, node_b, /)	Remove an edge between 2 nodes.
remove_edge_from_index(edge, /)	Remove an edge identified by the provided index
remove_edges_from(index_list, /)	Remove edges from the graph.
remove_node(node, /)	Remove a node from the graph.
remove_nodes_from(index_list, /)	Remove nodes from the graph.
subgraph(nodes, /)	Return a new PyGraph object for a subgraph of this graph
to_dot([node_attr, edge_attr, graph_attr, ...])	Generate a dot file from the graph
update_edge(source, target, /, edge)	Update an edge's weight/payload in place
update_edge_by_index(source, target, /, edge)	Update an edge's weight/data payload in place by the edge index

continues on next page

Table 2 – continued from previous page

weighted_edge_list()	Get edge list with weights

Attributes

multigraph	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 outbound) 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 an node from other onto this graph.
- **edge_map_func** – An optional python callabble 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

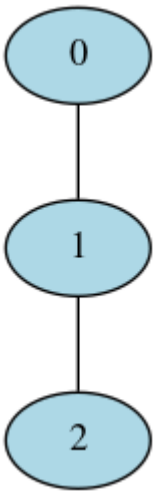
For example, start by building a graph:

```
import os
import tempfile
```

(continues on next page)

(continued from previous page)

```
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
```



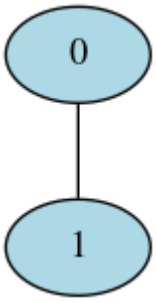
Then build a second one:

```
# 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(
    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)
```

(continues on next page)

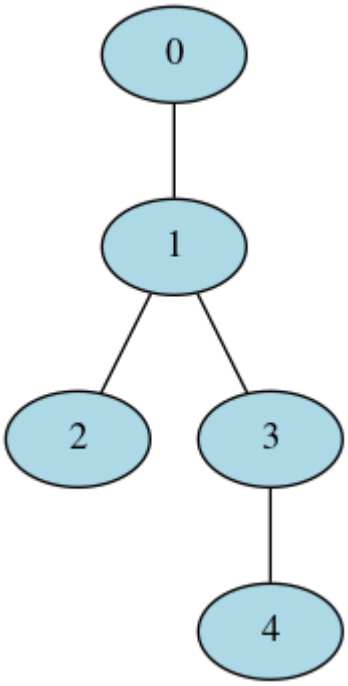
(continued from previous page)

```
    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_lsit*, */*)
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 with return a list of neighbor node indices

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

Returns A list of the neighbor node indicies

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*, *deliminator=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 seperated list of deliminated 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
- **deliminator** (*str*) – Optional character to use as a deliminator by default any whitespace will be used

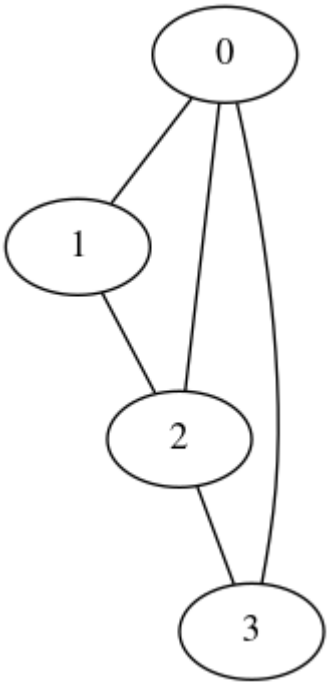
For example:

```
import os
import tempfile

from PIL import Image
import pydot

import retworkx


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 = retworkx.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')
```

(continues on next page)

(continued from previous page)

```
    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 indicies 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 retworkx 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 retworkx 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 retworkx 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

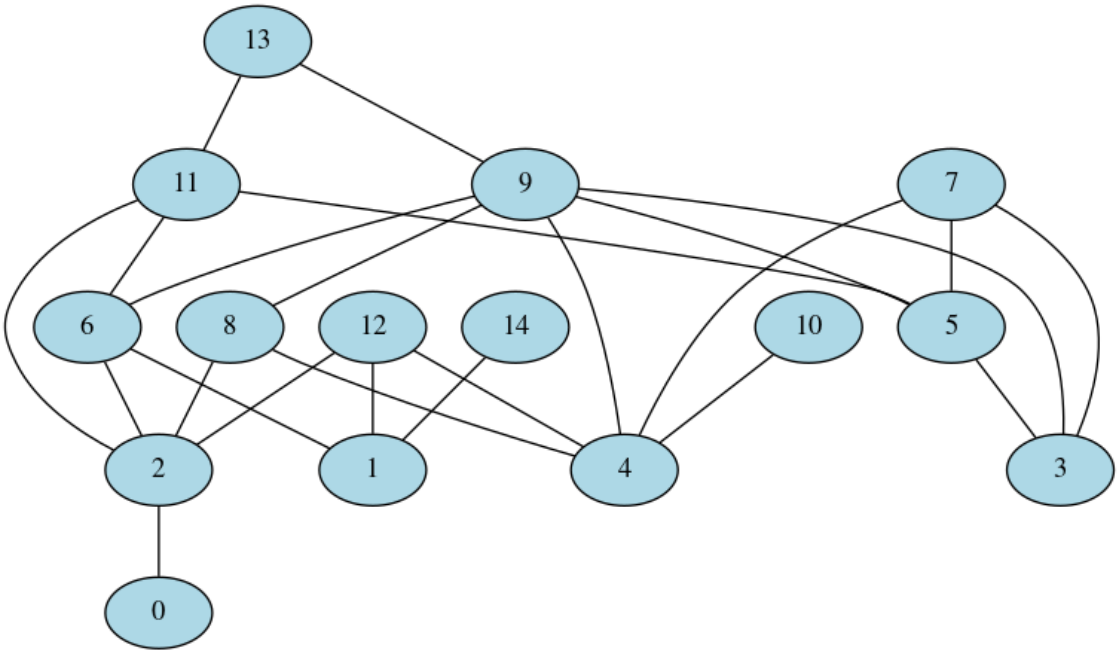
Using this method enables you to leverage graphviz to visualize a *retworkx.PyGraph* object. For example:

```
import os
import tempfile

import pydot
from PIL import Image

import retworkx

graph = retworkx.undirected_gnp_random_graph(15, .25)
dot_str = graph.to_dot(
    lambda node: dict(
        color='black', fillcolor='lightblue', style='filled'))
```

(continues on next page)

(continued from previous page)

```
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*

2.1.2 retworkx.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 retworkx

graph = retworkx.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 retworkx

graph = retworkx.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 retworkx
dag = retworkx.PyDiGraph()
dag.check_cycle = True
```

or at object creation:

```
import retworkx
dag = retworkx.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

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

continues on next page

Table 4 – continued from previous page

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

Attributes

check_cycle	Whether cycle checking is enabled for the DiGraph/DAG.
multigraph	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 dat 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")
}
```

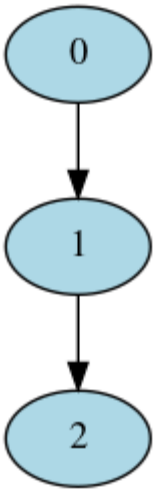
- **node_map_func** – An optional python callable that will take in a single node weight/data object and return a new node weight/data object that will be used when adding an node from other onto this graph.
- **edge_map_func** – An optional python callable that will take in a single edge weight/data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

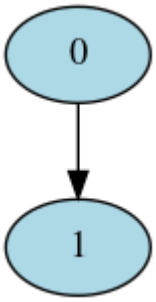
For example, start by building a graph:

```
import os
import tempfile

import pydot
from PIL import Image

import retworkx

# Build first graph and visualize:
graph = retworkx.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 = retworkx.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
```



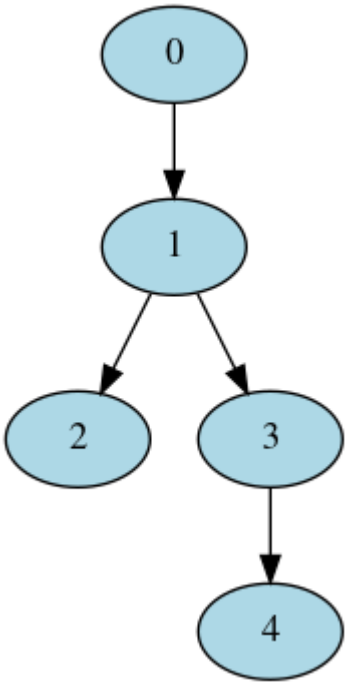
Finally compose the `other_graph` onto `graph`

```
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_lsit*, */*)
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 a 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 indicies 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 indicies

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*, *deliminator=None*)

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

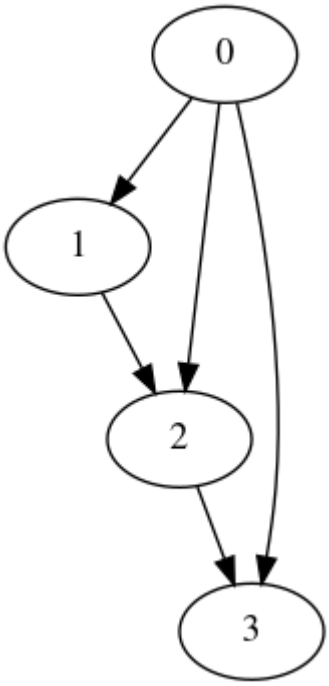
The expected format for the edge list file is a line seperated list of deliminated node ids. If there are more than 3 elements on a line the 3rd on will be treated as a string weight for the edge

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
- **deliminator** (*str*) – Optional character to use as a deliminator by default any whitespace will be used

For example:

```
import os
import tempfile

from PIL import Image
import pydot

import retworkx


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 = retworkx.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 ingored and this function willl 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 indicies 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 indicies for the succesors of a node

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

Returns A list of the neighbor node indicies

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

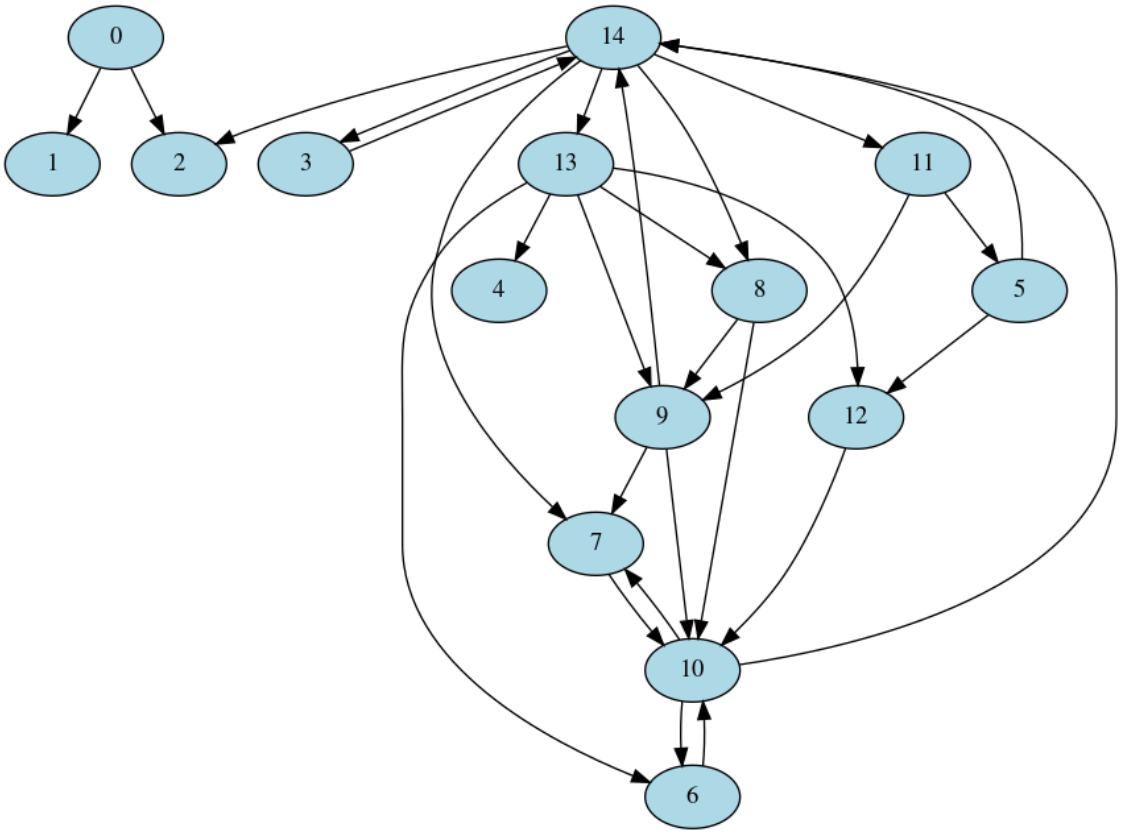
- **edge_attr** – A callable that will take in an edge data object and return a dictionary of attributes to be associated with the node in the dot file. The key and value of this dictionary **must** be a string. If they're not strings retworkx will raise TypeError (unfortunately without an error message 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 retworkx 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 *retworkx.PyDiGraph* object. For example:

```
import os
import tempfile

import pydot
from PIL import Image

import retworkx

graph = retworkx.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*

2.1.3 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 interchangably 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 retworkx
dag = retworkx.PyDAG()
dag.check_cycle = True
```

or at object creation:

```
import retworkx
dag = retworkx.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

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

continues on next page

Table 6 – continued from previous page

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

Attributes

check_cycle	Whether cycle checking is enabled for the DiGraph/DAG.
multigraph	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 dat 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")
}
```

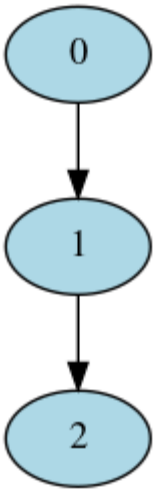
- **node_map_func** – An optional python callable that will take in a single node weight/data object and return a new node weight/data object that will be used when adding an node from other onto this graph.
- **edge_map_func** – An optional python callable that will take in a single edge weight/data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

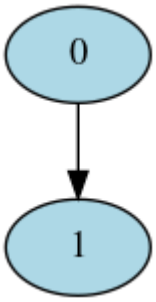
For example, start by building a graph:

```
import os
import tempfile

import pydot
from PIL import Image

import retworkx

# Build first graph and visualize:
graph = retworkx.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
```



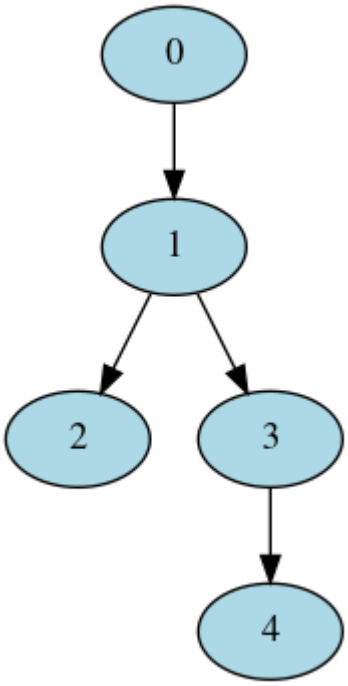
Finally compose the `other_graph` onto `graph`

```
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_lsit*, /)
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 a 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 indicies 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 indicies

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*, *deliminator=None*)

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

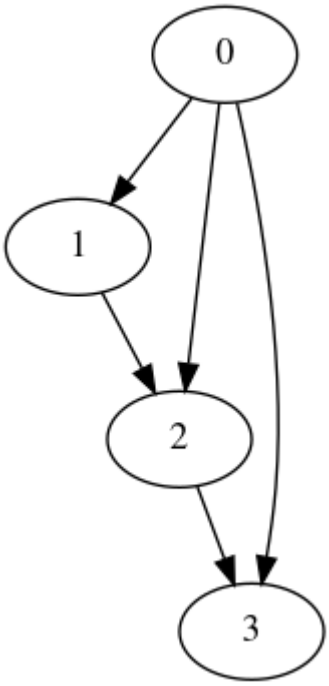
The expected format for the edge list file is a line seperated list of deliminated node ids. If there are more than 3 elements on a line the 3rd on will be treated as a string weight for the edge

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
- **deliminator** (*str*) – Optional character to use as a deliminator by default any whitespace will be used

For example:

```
import os
import tempfile

from PIL import Image
import pydot

import retworkx


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 = retworkx.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 ingored and this function willl 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 indicies 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 indicies for the succesors of a node

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

Returns A list of the neighbor node indicies

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

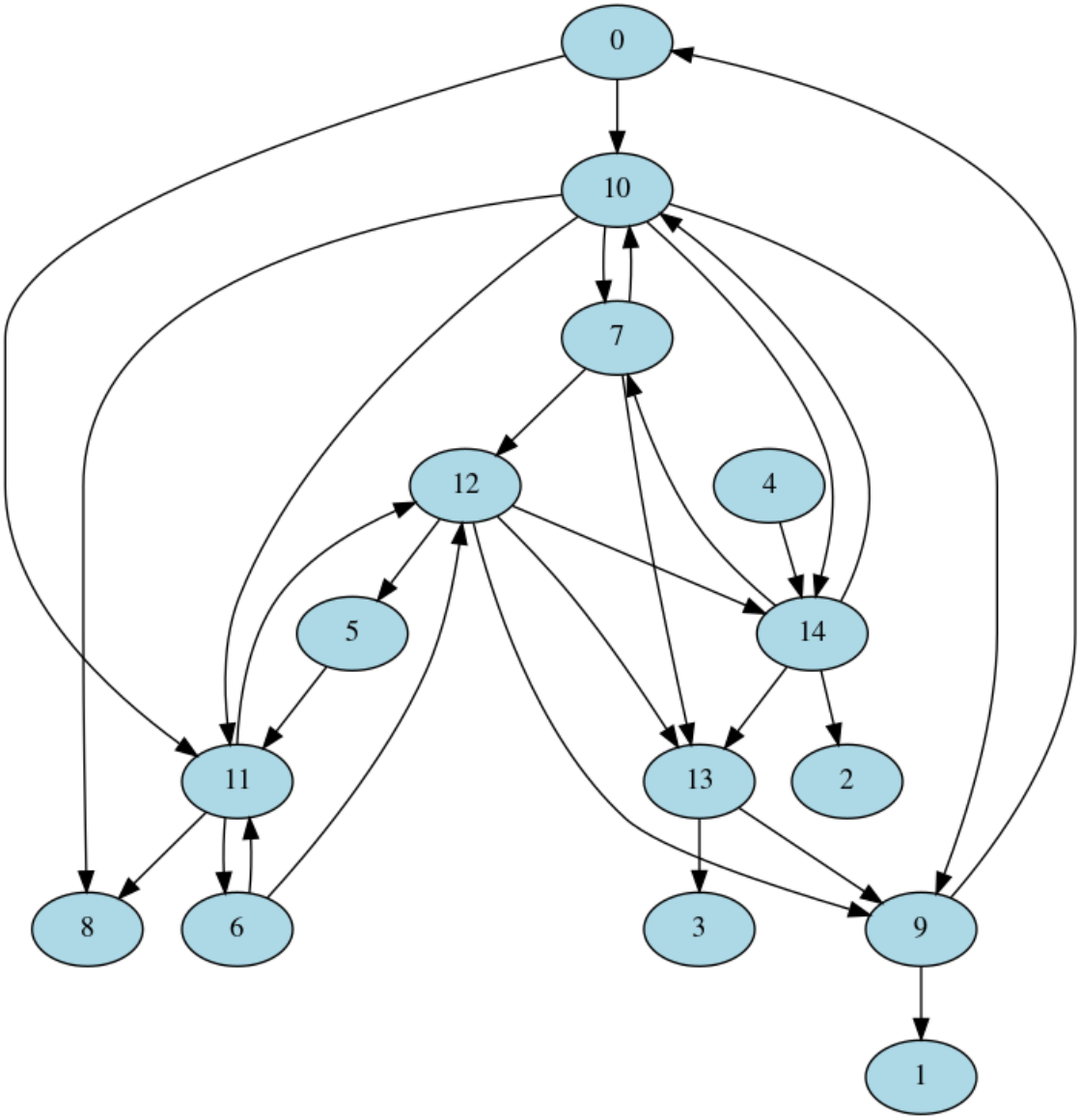
- **edge_attr** – A callable that will take in an edge data object and return a dictionary of attributes to be associated with the node in the dot file. The key and value of this dictionary **must** be a string. If they're not strings networkx will raise TypeError (unfortunately without an error message 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

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

2.2.1 networkx.generators.cycle_graph

cycle_graph(*num_nodes=None*, *weights=None*, *multigraph=True*, */*)

Generate an undirected cycle graph

Parameters

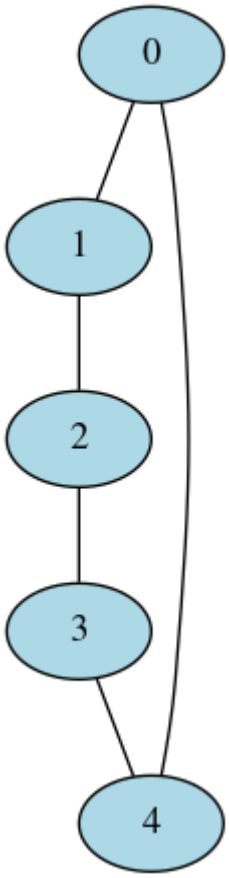
- **num_node** (*int*) – The number of nodes to generate the graph with. Node weights will be None if this is specified. If both `num_node` and `weights` are set this will be ignored and `weights` will be used.
- **weights** (*list*) – A list of node weights, the first element in the list will be the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

graph = retworkx.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
```



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*

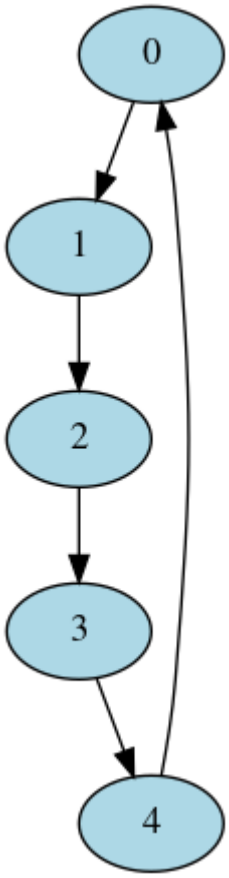
Raises **IndexError** – If neither `num_nodes` or `weights` are specified

```
import os
import tempfile
```

(continues on next page)

(continued from previous page)

```
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
```



2.2.3 networkx.generators.path_graph

path_graph(*num_nodes=None*, *weights=None*, *multigraph=True*, */*)

Generate an undirected path graph

Parameters

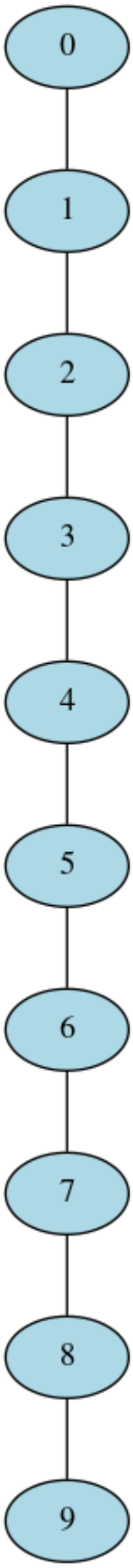
- **num_node** (*int*) – The number of nodes to generate the graph with. Node weights will be None if this is specified. If both `num_node` and `weights` are set this will be ignored and `weights` will be used.
- **weights** (*list*) – A list of node weights, the first element in the list will be the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


0
1
2
3
4
5
6
7
8
9

2.2.4 retworkx.generators.directed_path_graph

directed_path_graph(*num_nodes=None*, *weights=None*, *bidirectional=False*, */*)

Generate a directed path graph

Parameters

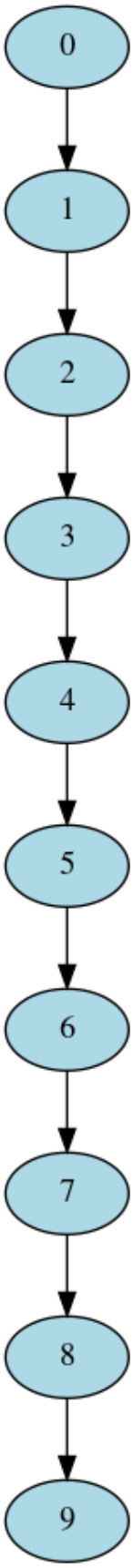
- **num_node** (*int*) – The number of nodes to generate the graph with. Node weights will be None if this is specified. If both `num_node` and `weights` are set this will be ignored and `weights` will be used.
- **weights** (*list*) – A list of node weights, the first element in the list will be the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

graph = retworkx.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
```


0
1
2
3
4
5
6
7
8
9

2.2.5 networkx.generators.star_graph

star_graph(*num_nodes=None*, *weights=None*, *multigraph=True*, */*)

Generate an undirected star graph

Parameters

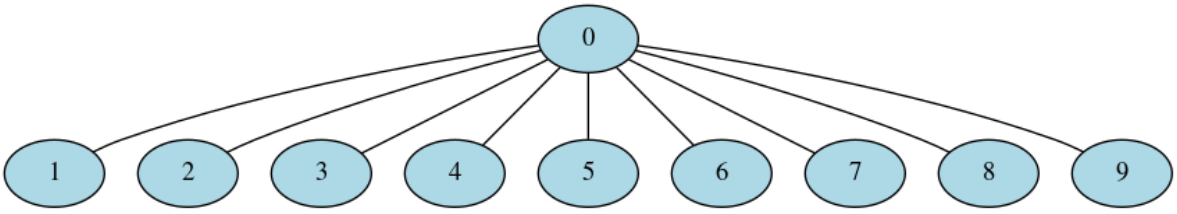
- **num_node** (*int*) – The number of nodes to generate the graph with. Node weights will be None if this is specified. If both `num_node` and `weights` are set this will be ignored and `weights` will be used.
- **weights** (*list*) – A list of node weights, the first element in the list will be the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



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

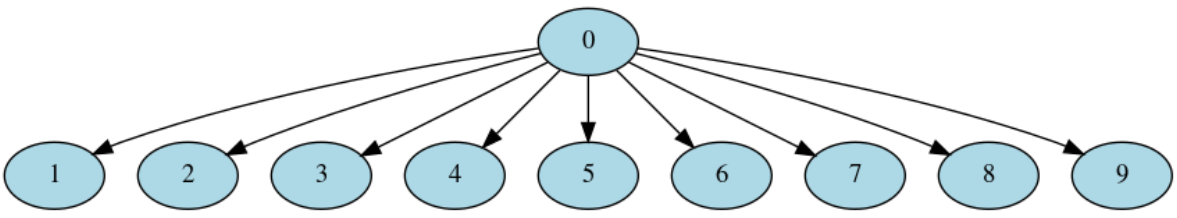
- **num_node** (*int*) – The number of nodes to generate the graph with. Node weights will be None if this is specified. If both `num_node` and `weights` are set this will be ignored and `weights` will be used.
- **weights** (*list*) – A list of node weights, the first element in the list will be the 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:
    tmp_path = os.path.join(tmpdirname, 'dag.png')
    dot.write_png(tmp_path)
    image = Image.open(tmp_path)
    os.remove(tmp_path)
image
```

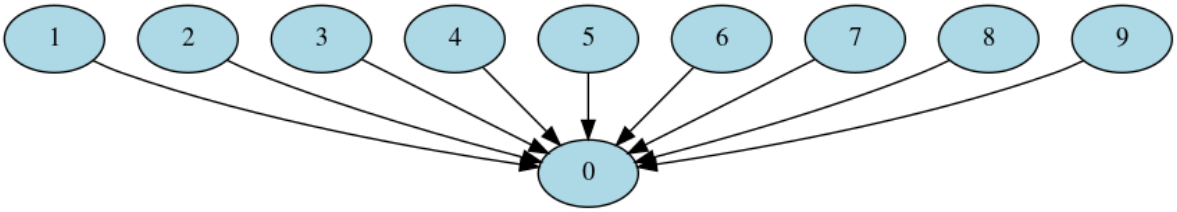



```
import os
import tempfile
```

(continues on next page)

(continued from previous page)

```
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
```



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*

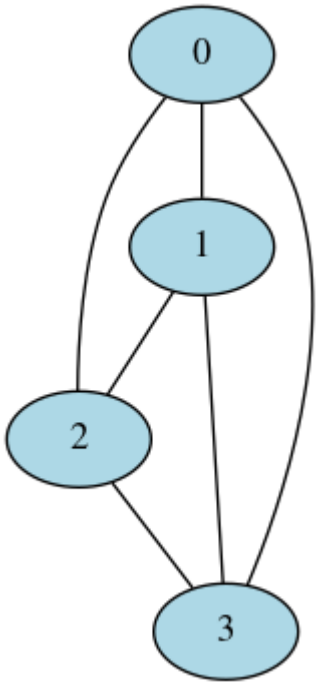
Raises **IndexError** – If neither `num_nodes` or `weights` are specified

```
import os
import tempfile

import pydot
```

(continues on next page)

(continued from previous page)

```
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
```



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

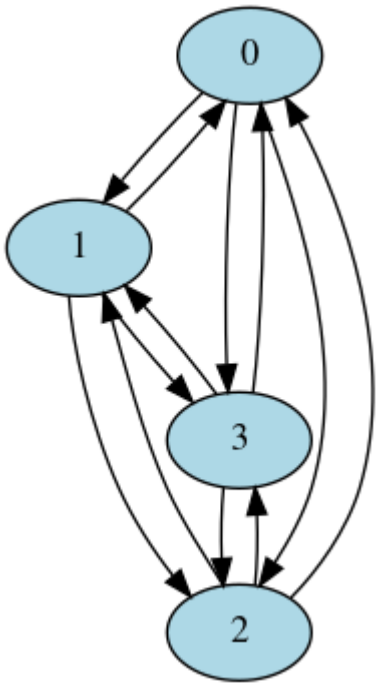
- **num_node** (*int*) – The number of nodes to generate the graph with. Node weights will be None if this is specified. If both `num_node` and `weights` are set this will be ignored and `weights` will be used.
- **weights** (*list*) – A list of node weights. If both `num_node` and `weights` are set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

import retworkx.generators

graph = retworkx.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
```



2.2.9 networkx.generators.grid_graph

grid_graph(*rows=None*, *cols=None*, *weights=None*, *multigraph=True*, */*)

Generate an undirected grid graph.

Parameters

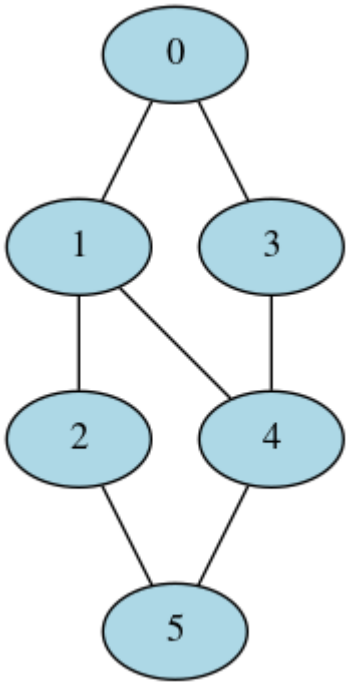
- **rows** (*int*) – The number of rows to generate the graph with. If specified, cols also need to be specified
- **cols** (*list*) – The number of rows to generate the graph with. If specified, rows also need to be specified. rows*cols defines the number of nodes in the graph
- **weights** (*list*) – 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 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 retworkx.generators

graph = retworkx.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
```



2.2.10 retworkx.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*

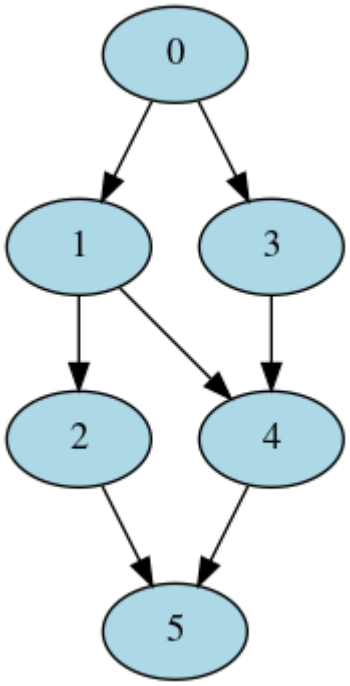
Raises **IndexError** – If neither `rows` or `cols` and `weights` are specified

```
import os
import tempfile
```

(continues on next page)

(continued from previous page)

```
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

networkx.directed_gnp_random_graph(...[, seed])	Return a G_{np} directed random graph, also known as an Erdős-Rényi graph or a binomial graph.
networkx.undirected_gnp_random_graph(...[, seed])	Return a G_{np} random undirected graph, also known as an Erdős-Rényi graph or a binomial graph.

continues on next page

Table 9 – continued from previous page

retworkx.directed_gnm_random_graph(...[, seed])	Return a G_{nm} of a directed graph
retworkx.undirected_gnm_random_graph(...[, seed])	Return a G_{nm} of an undirected graph

2.3.1 retworkx.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[1] 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`[2]

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*

2.3.2 retworkx.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[1] 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`[2]

Parameters

- **num_nodes** (*int*) – The number of nodes to create in the graph

[1] Vladimir Batagelj and Ulrik Brandes, "Efficient generation of large random networks", Phys. Rev. E, 71, 036113, 2005.
[2] https://github.com/networkx/networkx/blob/networkx-2.4/networkx/generators/random_graphs.py#L49-L120
[1] Vladimir Batagelj and Ulrik Brandes, "Efficient generation of large random networks", Phys. Rev. E, 71, 036113, 2005.
[2] https://github.com/networkx/networkx/blob/networkx-2.4/networkx/generators/random_graphs.py#L49-L120

- **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*

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 $m >= n(n-1)$ 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

Returns A PyDiGraph object

Return type *PyDiGraph*

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 $m >= n(n-1)/2$ 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

retworkx.bfs_successors(graph, node, /)	Return successors in a breadth-first-search from a source node.
retworkx.dag_longest_path(graph, /)	Find the longest path in a DAG
retworkx.dag_longest_path_length(graph, /)	Find the length of the longest path in a DAG
retworkx.number_weakly_connected_components(...)	Find the number of weakly connected components in a DAG.
retworkx.weakly_connected_components(graph, /)	Find the weakly connected components in a directed graph
retworkx.is_weakly_connected(graph, /)	Check if the graph is weakly connected
retworkx.is_directed_acyclic_graph(graph, /)	Check that the PyDiGraph or PyDAG doesn't have a cycle
retworkx.is_isomorphic(first, second, /)	Determine if 2 graphs are structurally isomorphic
retworkx.is_isomorphic_node_match(first, ...)	Determine if 2 DAGs are isomorphic
retworkx.topological_sort(graph, /)	Return the topological sort of node indexes from the provided graph
retworkx.descendants(graph, node, /)	Return the descendants of a node in a graph.
retworkx.ancestors(graph, node, /)	Return the ancestors of a node in a graph.
retworkx.lexicographical_topological_sort(...)	Get the lexicographical topological sorted nodes from the provided DAG
retworkx.graph_distance_matrix(graph, /[, ...])	Get the distance matrix for an undirected graph
retworkx.digraph_distance_matrix(graph, /[, ...])	Get the distance matrix for a directed graph
retworkx.floyd_warshall(dag, /)	Return the shortest path lengths between ever pair of nodes that has a path connecting them
retworkx.graph_floyd_warshall_numpy(graph, /)	Find all-pairs shortest path lengths using Floyd's algorithm
retworkx.digraph_floyd_warshall_numpy	Find all-pairs shortest path lengths using Floyd's algorithm
retworkx.collect_runs(graph, filter)	Collect runs that match a filter function
retworkx.layers(dag, first_layer, /)	Return a list of layers
retworkx.digraph_adjacency_matrix(graph, /)	Return the adjacency matrix for a PyDiGraph object
retworkx.graph_adjacency_matrix(graph, /[, ...])	Return the adjacency matrix for a PyGraph class
retworkx.graph_all_simple_paths	Return all simple paths between 2 nodes in a PyGraph object
retworkx.digraph_all_simple_paths	Return all simple paths between 2 nodes in a PyDiGraph object
retworkx.graph_astar_shortest_path(graph, ...)	Compute the A* shortest path for a PyGraph
retworkx.digraph_astar_shortest_path(graph, ...)	Compute the A* shortest path for a PyDiGraph
retworkx.graph_dijkstra_shortest_paths	Find the shortest path from a node
retworkx.digraph_dijkstra_shortest_paths	Find the shortest path from a node
retworkx.graph_dijkstra_shortest_path_lengths(...)	Compute the lengths of the shortest paths for a PyGraph object using Dijkstra's algorithm
retworkx.digraph_dijkstra_shortest_path_lengths(...)	Compute the lengths of the shortest paths for a PyDiGraph object using Dijkstra's algorithm

continues on next page

Table 10 – continued from previous page

networkx.graph_k_shortest_path_lengths(...)	Compute the length of the kth shortest path
networkx.digraph_k_shortest_path_lengths(...)	Compute the length of the kth shortest path
networkx.graph_greedy_color(graph, /)	Color a PyGraph using a largest_first strategy greedy graph coloring.
networkx.cycle_basis(graph, /[, root])	Return a list of cycles which form a basis for cycles of a given PyGraph
networkx.strongly_connected_components(graph, /)	Compute the strongly connected components for a directed graph
networkx.graph_dfs_edges(graph, /[, source])	Get edge list in depth first order
networkx.digraph_dfs_edges(graph, /[, source])	Get edge list in depth first order
networkx.digraph_find_cycle(graph, /[, source])	Return the first cycle encountered during DFS of a given PyDiGraph, empty list is returned if no cycle is found
networkx.digraph_union(first, second, ...)	Return a new PyDiGraph by forming a union from two input PyDiGraph objects
networkx.is_matching(graph, matching, /)	Check if matching is valid for graph
networkx.is_maximal_matching(graph, matching, /)	Check if a matching is a maximal (**not** maximum) matching for a graph
networkx.max_weight_matching(graph, /[, ...])	Compute a maximum-weighted matching for a *PyGraph*

networkx.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*

networkx.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

retworkx.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

retworkx.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

retworkx.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 it a weakly connected component of the graph

Return type list

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

networkx.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

networkx.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

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

retworkx.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

retworkx.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

retworkx.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

retworkx.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

retworkx.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 `paralllel_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

retworkx.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

retworkx.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 itermediary 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

retworkx.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 retworkx/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

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 retworkx/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

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

retworkx.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

retworkx.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 retworkx/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

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

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

retworkx.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

retworkx.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*

retworkx.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*

retworkx.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

retworkx.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

retworkx.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

retworkx.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

networkx.graph_k_shortest_path_lengths

graph_k_shortest_path_lengths(*graph*, *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** (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

networkx.digraph_k_shortest_path_lengths

digraph_k_shortest_path_lengths(*graph*, *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** (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

retworkx.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

retworkx.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[1].

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

retworkx.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

retworkx.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 arbitrarly and repeated until all components of the graph are searched.

[1] Paton, K. An algorithm for finding a fundamental set of cycles of a graph. Comm. ACM 12, 9 (Sept 1969), 514-518.

Returns A list of edges as a tuple of the form `(source, target)` in depth-first order

Return type *EdgeList*

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 arbitrarly 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*

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*

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

retworkx.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

retworkx.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

retworkx.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[1].

Parameters

- **graph** (PyGraph) – The undirected graph to compute the max weight matching for. Expects to have no parallel edges (multigraphs are untested currently).
- **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 ofthe 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.

retworkx.distance_matrix()	Get the distance matrix for a graph
retworkx.floyd_warshall_numpy()	Return the adjacency matrix for a graph object
retworkx.adjacency_matrix()	Return the adjacency matrix for a graph object
retworkx.all_simple_paths()	Return all simple paths between 2 nodes in a PyGraph object
retworkx.astar_shortest_path()	Compute the A* shortest path for a graph
retworkx.dijkstra_shortest_paths()	Find the shortest path from a node
retworkx.dijkstra_shortest_path_lengths()	Compute the lengths of the shortest paths for a graph object using Dijkstra's algorithm.
retworkx.k_shortest_path_lengths()	Compute the length of the kth shortest path
retworkx.dfs_edges()	Get edge list in depth first order

[1] "Efficient Algorithms for Finding Maximum Matching in Graphs", Zvi Galil, ACM Computing Surveys, 1986.

retworkx.distance_matrix

distance_matrix(*graph*, *parallel_threshold=300*)
distance_matrix(*graph:* retworkx.PyDiGraph, *parallel_threshold=300*, *as_undirected=False*)
distance_matrix(*graph:* retworkx.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

retworkx.floyd_warshall_numpy

floyd_warshall_numpy(*graph*, *weight_fn=None*, *default_weight=1.0*)
floyd_warshall_numpy(*graph:* retworkx.PyDiGraph, *weight_fn=None*, *default_weight=1.0*)
floyd_warshall_numpy(*graph:* retworkx.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 retworkx/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

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 retworkx/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

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:*~retworkx.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

retworkx.astar_shortest_path

astar_shortest_path(*graph*, *node*, *goal_fn*, *edge_cost_fn*, *estimate_cost_fn*)
astar_shortest_path(*graph:* retworkx.PyDiGraph, *node*, *goal_fn*, *edge_cost_fn*, *estimate_cost_fn*)
astar_shortest_path(*graph:* retworkx.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*

retworkx.dijkstra_shortest_paths

dijkstra_shortest_paths(*graph*, *source*, *target=None*, *weight_fn=None*, *default_weight=1.0*, *as_undirected=False*)
dijkstra_shortest_paths(*graph:* retworkx.PyDiGraph, *source*, *target=None*, *weight_fn=None*, *default_weight=1.0*, *as_undirected=False*)
dijkstra_shortest_paths(*graph:* retworkx.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

retworkx.dijkstra_shortest_path_lengths

dijkstra_shortest_path_lengths(*graph*, *node*, *edge_cost_fn*, *goal=None*)
dijkstra_shortest_path_lengths(*graph:* retworkx.PyDiGraph, *node*, *edge_cost_fn*, *goal=None*)
dijkstra_shortest_path_lengths(*graph:* retworkx.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

retworkx.k_shortest_path_lengths

k_shortest_path_lengths(*graph*, *start*, *k*, *edge_cost*, *goal=None*)
k_shortest_path_lengths(*graph:* retworkx.PyDiGraph, *start*, *k*, *edge_cost*, *goal=None*)
k_shortest_path_lengths(*graph:* retworkx.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

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 arbitrarly 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

2.5.1 networkx.InvalidNode

exception InvalidNode

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

2.5.2 networkx.DAGWouldCycle

exception DAGWouldCycle

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

2.5.3 networkx.NoEdgeBetweenNodes

exception NoEdgeBetweenNodes

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

2.5.4 networkx.DAGHasCycle

exception DAGHasCycle

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

2.5.5 networkx.NoSuitableNeighbors

exception NoSuitableNeighbors

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

2.5.6 networkx.NoPathFound

exception NoPathFound

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

2.5.7 networkx.NullGraph

exception NullGraph

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

2.6 Return Iterator Types

networkx.BFSSuccessors	A custom class for the return from *networkx.bfs_successors()*
networkx.NodeIndices	A custom class for the return of node indices
networkx.EdgeList	A custom class for the return of edge lists
networkx.WeightedEdgeList	A custom class for the return of edge lists with weights

2.6.1 networkx.BFSSuccessors

class BFSSuccessors

A custom class for the return from *networkx.bfs_successors()*

This class is a container class for the results of the *networkx.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 networkx

graph = networkx.generators.directed_path_graph(5)
bfs_succ = networkx.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

__init__()	Initialize self.

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
```

(continues on next page)

(continued from previous page)

```
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(node)
first_element = next(nodes_iter)
second_element = next(nodes_iter)
```

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

Methods

__init__()	Initialize self.

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.

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]
# 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.

CHAPTER

THREE

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

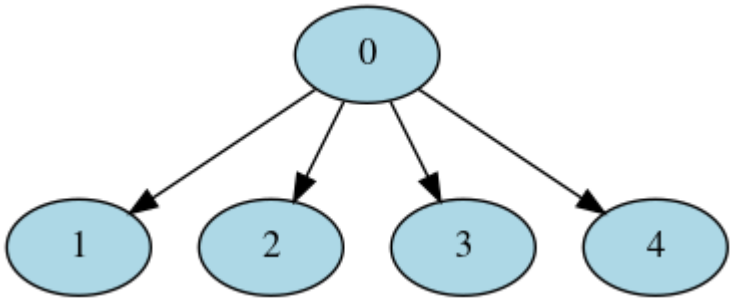
- A new constructor method *from_adjacency_matrix()* has been added to the *PyDiGraph* and *PyGraph* (*from_adjacency_matrix()*) classes. It enables creating a new graph from an input adjacency_matrix. For example:

```
import os
import tempfile

import numpy as np
import pydot
from PIL import Image

import retworkx


# 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 = retworkx.PyDiGraph.from_adjacency_matrix(adjacency_matrix)
# Draw graph
dot_str = graph.to_dot(
```

(continues on next page)

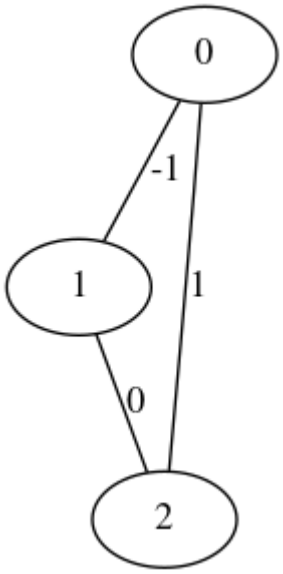
(continued from previous page)

```
    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
```



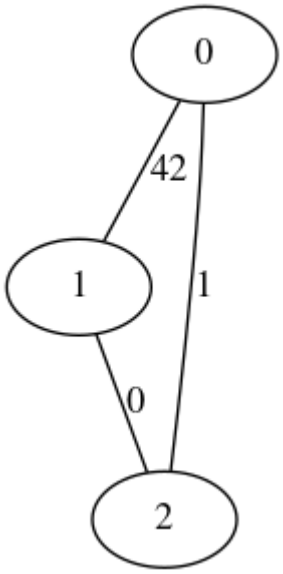
- A new algorithm function, *is_matching()*, was added to check if a matching set is valid for given *PyGraph* object.
- A new algorithm function, `is_maxmimal_matching()`, was added to check if a matching set is valid and maximal for a given *PyGraph* object.
- Add a new function, *max_weight_matching()* 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 retworkx as rx

graph = rx.PyGraph(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 `retworkx.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 *retworkx.PyDiGraph* and *retworkx.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 *retworkx.PyDiGraph.neighbors()* and *retworkx.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.

CHAPTER

FOUR

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.

CHAPTER

FIVE

0.7.0

This release includes several new features and bug fixes.

This release also dropped support for Python 3.5. If you want to use retworkx 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 `retworkx.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, *retworkx.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 retworkx on Python 3.5 has been dropped. The last release with support for Python 3.5 is 0.6.0.
- The *retworkx.PyDiGraph.node_indexes()*, *retworkx.PyDiGraph.neighbors()*, *retworkx.PyDiGraph.successor_indices()*, *retworkx.PyDiGraph.predecessor_indices()*, *retworkx.PyDiGraph.add_nodes_from()*, *retworkx.PyGraph.node_indexes()*, *retworkx.PyGraph.add_nodes_from()*, and *retworkx.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 *retworkx.PyDiGraph.edge_list()*, and *retworkx.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 *retworkx.PyDiGraph.weighted_edge_list()*, *retworkx.PyDiGraph.in_edges()*, *retworkx.PyDiGraph.out_edges()*, and *retworkx.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 retworkx 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.

CHAPTER

SIX

0.6.0

This release includes a number of new features and bug fixes. The main focus of this release was to expand the retworkx 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 retworkx 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

CHAPTER

SEVEN

0.5.0

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 Graphivz (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, `retworkx.generators`, has been added. Functions in this module can be used for quickly generating graphs of certain shape. To start it includes:
  - *retworkx.generators.cycle_graph()*
  - *retworkx.generators.directed_cycle_graph()*
  - *retworkx.generators.path_graph()*
  - *retworkx.generators.directed_path_graph()*
  - *retworkx.generators.star_graph()*
  - *retworkx.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 predecesors 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.

CHAPTER

EIGHT

0.4.0

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 retworkx 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 it's 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 retworkx. 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 retworkx exception classes are now properly exported from the retworkx 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 retworkx exception class can be used.

CHAPTER

NINE

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

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 yaml file in releasenotes/notes. Then open that yaml file in a text editor and write the release note. The basic structure of a release note is restructured text in yaml 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:`~retworkx.foo` that adds support for doing
    something to :class:`~retworkx.PyDiGraph` objects.
  - |
    The :class:`~retworkx.PyDiGraph` class has a new method
    :meth:`~retworkx.PyDiGraph.foo``. This is the equivalent of calling the
    :func:`~retworkx.foo` function to do something to your
    :class:`~retworkx.PyDiGraph` object, but provides the convenience of running
    it natively on an object. For example::

      from retworkx import PyDiGraph

      g = PyDiGraph.
      g.foo()

deprecations:
  - |
    The ``retworkx.bar`` function has been deprecated and will be removed in a
    future release. It has been superseded by the
    :meth:`~retworkx.PyDiGraph.foo` method and :func:`~retworkx.foo` function
    which provides similar functionality but with more accurate results and
    better performance. You should update your calls
    ``retworkx.bar()`` calls to use ``retworkx.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/retworkx/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 networkx 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 networkx 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`

CHAPTER

TEN

RETWORKX FOR NETWORKX USERS

This is an introductory guide for existing networkx users on how to use retworkx, how it differs from networkx, and key differences to keep in mind.

10.1 Some Key Differences

retworkx (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 retworkx being written in Rust puts more constraints on how you interact with graph objects. With retworkx 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 retworkx the above example would be something like:

```
import retworkx 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 retworkx 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 retworkx.

Similarly when there are algorithm functions that operate on a node or edge data, callback functions are used in retworkx 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 retworkx you would use:

```
import retworkx 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 retworkx 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 retworkx 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 retworkx 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 retworkx 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. Retworkx 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 retworkx 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.

retworkx 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 retworkx 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	retworkx	Notes
`Graph()`	*PyGraph(multigraph=False)*	Only in multigraph flag added in retworkx>= 0.8.0 prior releases always allow multiple edges
`DiGraph()`	*PyDiGraph(multigraph=False)*	Only in multigraph flag added in retworkx>= 0.8.0 prior releases always allow multiple edges
`MultiGraph()`	*PyGraph()*	
`MultiDiGraph()`	*PyDiGraph()*	

The other thing to note here is that retworkx 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	retworkx	Notes
`Graph([(0, 1), (1, 0)])`	`graph = PyGraph()` `graph.extend_from_edge_list([(0, 1), (1, 0)])`	retworkx input must be a list of 2-tuples, while networkx can be an iterator
`Graph([(0, 1, {'weight': 2}), (1, 0, {'weight': 1})])`	`graph = PyGraph()` `graph.extend_from_edge_list([(0, 1, 2), (1, 0, 1)])`	retworkx input must be a list of 3-tuples, while networkx can be an iterator
`Graph(np.array([[0, 1, 1], [1, 0, 1], [1, 0, 1]]))`	`PyGraph.from_adjacency_matrix(np.array([[0, 1, 1], [1, 0, 1], [1, 0, 1]], dtype=np.float64))`	retworkx *from_adjacency_matrix()* can only take a float dtype numpy array, you can use `.astype(np.float64, copy=False)` to adapt a non-float array.

10.3.2 Graph Modifiers

networkx	retworkx	Notes
`add_node()`	*add_node()*	retworkx returns a node index for the newly created node
`add_nodes_from()`	*add_nodes_from()*	retworkx requires the input to be a list of objects and will return a list of node indices for the newly created nodes
`add_edge()`	*add_edge()*	retworkx requires 3 parameters be used, the 2 node indices and the payload (networkx works with either 2 or 3)
`add_edges_from()`	*add_edges_from()*, *add_edges_from_no_data()*, *extend_from_edge_list()*, *extend_from_weighted_edge_list()*	retworkx requires a list of either a 3 or 2 tuple (depending on whether weights/data are expected or not). The difference between the retworkx `extend_from*` and `add_edges_from*` methods are that the `extend_from*` will create new nodes with a weight/data payload of `None` if any node indices are missing.

(note the retworkx 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.

INDEX

Symbols

A

B

C

D

E

F

G

H

I

W