
PyGraphviz Documentation

Release 1.10rc1.dev0

PyGraphviz Developers

May 31, 2022

CONTENTS

1	Install	1
1.1	Recommended	1
1.2	Advanced	2
1.3	FAQ	4
2	Tutorial	5
2.1	Start-up	5
2.2	Graphs	5
2.3	Nodes, and edges	6
2.4	Attributes	6
2.5	Layout and Drawing	7
3	Gallery	9
3.1	Subgraph	9
3.2	Attributes	10
3.3	Basic	11
3.4	Star	12
3.5	Knuth Miles	15
4	Reference	19
4.1	AGraph Class	19
4.2	FAQ	33
4.3	API Notes	34
4.4	News	35
4.5	Related Packages	39
4.6	History	39
4.7	Credits	39
4.8	Legal	40
5	Contributor Guide	41
5.1	Development Workflow	41
5.2	Divergence from <code>upstream main</code>	43
5.3	Guidelines	44
5.4	Testing	44
5.5	Adding tests	44
5.6	Adding examples	45
5.7	Bugs	45
	Index	47

INSTALL




PyGraphviz requires:

- Python (version 3.8, 3.9, or 3.10)
- [Graphviz](#) (version 2.42 or later)
- C/C++ Compiler

Note: These instructions assume you have Python and a C/C++ Compiler on your computer.

Warning: Do not use the default channels to install pygraphviz with `conda`. The conda-forge channel should be used instead:

```
conda install --channel conda-forge pygraphviz
```

-  conda-forge-windows
-  conda-forge-windows
-  conda-forge-windows

1.1 Recommended

We recommend installing Python packages using [pip](#) and [virtual environments](#).

1.1.1 Linux

We recommend installing Graphviz using your Linux system's package manager. Below are examples for some popular distributions.

Ubuntu and Debian

```
$ sudo apt-get install graphviz graphviz-dev
$ pip install pygraphviz
```

Fedora and Red Hat

You may need to replace `dnf` with `yum` in the example below.

```
$ sudo dnf install graphviz graphviz-devel
$ pip install pygraphviz
```

1.1.2 macOS

We recommend installing Graphviz using the Homebrew package manager or MacPorts for macOS.

Homebrew

```
$ brew install graphviz
$ pip install pygraphviz
```

MacPorts

```
$ port install graphviz
$ pip install pygraphviz
$ pip install --global-option=build_ext \
    --global-option="-I/opt/local/include/" \
    --global-option="-L/opt/local/lib/" \
    pygraphviz
```

1.2 Advanced

The two main difficulties are

1. installing Graphviz and
2. informing pip where Graphviz is installed.

1.2.1 Providing path to Graphviz

If you've installed Graphviz and `pip` is unable to find Graphviz, then you need to provide `pip` with the path(s) where it can find Graphviz. To do this, you first need to figure out where the binary files, includes files, and library files for Graphviz are located on your file system.

Once you know where you've installed Graphviz, you will need to do something like the following. There is an additional example using Chocolatey on Windows further down the page.

1.2.2 Windows

Historically, installing Graphviz and PyGraphviz on Windows has been challenging. Fortunately, the Graphviz developers are working to fix this and their recent releases have much improved the situation.

For this reason, PyGraphviz 1.7 only supports Graphviz 2.46.0 or higher on Windows. We recommend either manually installing the official binary release of Graphviz or using [Chocolatey](#), which has been updated to Graphviz 2.46.0.

You may also need to install Visual C/C++, e.g. from here: <https://visualstudio.microsoft.com/visual-cpp-build-tools/>

Assuming you have Python and Visual C/C++ installed, we believe the following should work on Windows 10 (64 bit) using PowerShell.

Manual download

1. Download and install 2.46.0 for Windows 10 (64-bit): [stable_windows_10_cmake_Release_x64_graphviz-install-2.46.0-win64.exe](#).
2. Install PyGraphviz via

```
PS C:\> python -m pip install --global-option=build_ext `
    --global-option="-IC:\Program Files\Graphviz\include" `
    --global-option="-LC:\Program Files\Graphviz\lib" `
    pygraphviz
```

Chocolatey

```
PS C:\> choco install graphviz
PS C:\> python -m pip install --global-option=build_ext `
    --global-option="-IC:\Program Files\Graphviz\include" `
    --global-option="-LC:\Program Files\Graphviz\lib" `
    pygraphviz
```

1.3 FAQ

Q

I followed the installation instructions but when I do:

```
>>> import pygraphviz
```

I get an error like:

```
ImportError: libagraph.so.1: cannot open shared object
file: No such file or directory
```

What is wrong?

A

Some Unix systems don't include the Graphviz library in the default search path for the run-time linker. The path is often something like `/usr/lib/graphviz` or `/sw/lib/graphviz` etc. and it needs to be added to your search path. On *nix systems, the preferred way to do this is by setting the appropriate flags when building/installing `pygraphviz`. For example, if the Graphviz libraries are installed in `/opt/lib/mygviz/` on your system:

```
pip install --global-option=build_ext \
            --global-option="-L/opt/lib/mygviz/" \
            --global-option="-R/opt/lib/mygviz/" \
            pygraphviz
```

In this example, the `-L` and `-R` flags tell the linker where to look for the required Graphviz libraries at build time and run time, respectively.

Q

How do I compile `pygraphviz` under Windows?

A

See [Windows](#) for the latest on how to install Graphviz and `pygraphviz` on Windows.

Q

Why don't you distribute a `pygraphviz` Windows installer?

A

We would very much like to make binary wheels available for `pygraphviz`, but there are several complications. `pygraphviz` is a wrapper around Graphviz, which means that Graphviz must be installed, and Graphviz header files, libraries *and* command line executables must all be accessible for the wrapper. The recommended use of the [Graphviz CLI](#) poses challenges for wheel packaging.

See also:

This [GitHub issue](#) for further discussion on wheels and packaging.

TUTORIAL

The API is very similar to that of NetworkX. Much of the NetworkX tutorial at <https://networkx.org/documentation/latest/tutorial.html> is applicable to PyGraphviz. See http://pygraphviz.github.io/documentation/latest/reference/api_notes.html for major differences.

2.1 Start-up

Import PyGraphviz with

```
>>> import pygraphviz as pgv
```

2.2 Graphs

To make an empty pygraphviz graph use the AGraph class:

```
>>> G = pgv.AGraph()
```

You can use the `strict` and `directed` keywords to control what type of graph you want. The default is to create a strict graph (no parallel edges or self-loops). To create a digraph with possible parallel edges and self-loops use

```
>>> G = pgv.AGraph(strict=False, directed=True)
```

You may specify a dot format file to be read on initialization:

```
>>> G = pgv.AGraph("Petersen.dot")
```

Other options for initializing a graph are using a string,

```
>>> G = pgv.AGraph("graph {1 - 2}")
```

using a dict of dicts,

```
>>> d = {"1": {"2": None}, "2": {"1": None, "3": None}, "3": {"2": None}}
>>> A = pgv.AGraph(d)
```

or using a SWIG pointer to the AGraph datastructure,

```
>>> h = A.handle
>>> C = pgv.AGraph(h)
```

2.3 Nodes, and edges

Nodes and edges can be added one at a time

```
>>> G.add_node("a")    # adds node 'a'
>>> G.add_edge("b", "c") # adds edge 'b'-'c' (and also nodes 'b', 'c')
```

or from lists or containers.

```
>>> nodelist = ["f", "g", "h"]
>>> G.add_nodes_from(nodelist)
```

If the node is not a string an attempt will be made to convert it to a string

```
>>> G.add_node(1)    # adds node '1'
```

2.4 Attributes

To set the default attributes for graphs, nodes, and edges use the `graph_attr`, `node_attr`, and `edge_attr` dictionaries

```
>>> G.graph_attr["label"] = "Name of graph"
>>> G.node_attr["shape"] = "circle"
>>> G.edge_attr["color"] = "red"
```

Graph attributes can be set when initializing the graph

```
>>> G = pgv.AGraph(ranksep="0.1")
```

Attributes can be added when adding nodes or edges,

```
>>> G.add_node(1, color="red")
>>> G.add_edge("b", "c", color="blue")
```

or through the node or edge attr dictionaries,

```
>>> n = G.get_node(1)
>>> n.attr["shape"] = "box"
```

```
>>> e = G.get_edge("b", "c")
>>> e.attr["color"] = "green"
```

2.5 Layout and Drawing

Pygraphviz provides several methods for layout and drawing of graphs.

To store and print the graph in dot format as a Python string use

```
>>> s = G.string()
```

To write to a file use

```
>>> G.write("file.dot")
```

To add positions to the nodes with a Graphviz layout algorithm

```
>>> G.layout()    # default to neato
>>> G.layout(prog="dot")    # use dot
```

To render the graph to an image

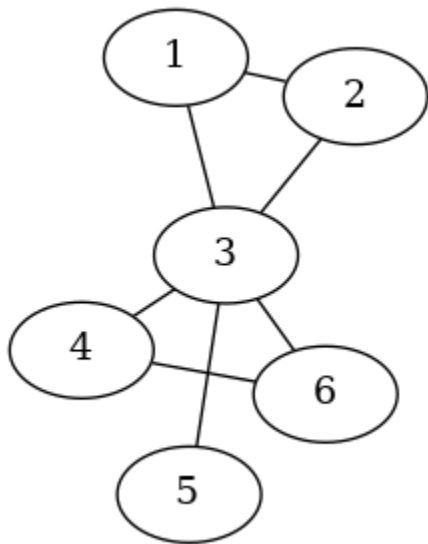
```
>>> G.draw("file.png")    # write previously positioned graph to PNG file
>>> G.draw("file.ps", prog="circo")    # use circo to position, write PS file
```


GALLERY

Graph visualization examples with pygraphviz.

3.1 Subgraph

Specify a subgraph in pygraphviz.



Out:

```
strict graph "" {
    subgraph s1 {
        graph [rank=same];
        4 -- 6;
        5;
    }
    1 -- 2;
    1 -- 3;
    2 -- 3;
    3 -- 4;
    3 -- 5;
    3 -- 6;
}
```

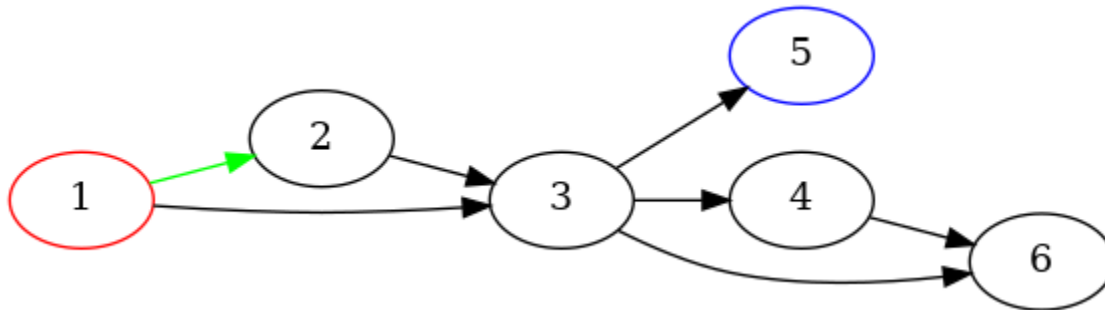
```
import pygraphviz as pgv

A = pgv.AGraph()
# add some edges
A.add_edge(1, 2)
A.add_edge(2, 3)
A.add_edge(1, 3)
A.add_edge(3, 4)
A.add_edge(3, 5)
A.add_edge(3, 6)
A.add_edge(4, 6)
# make a subgraph with rank='same'
B = A.add_subgraph([4, 5, 6], name="s1", rank="same")
B.graph_attr["rank"] = "same"
print(A.string()) # print dot file to standard output
A.draw("subgraph.png", prog="neato")
```

Total running time of the script: (0 minutes 0.239 seconds)

3.2 Attributes

Example illustrating how to set node, edge, and graph attributes for visualization.



Out:

```
strict digraph "" {
    graph [epsilon=0.001,
        rankdir=LR
    ];
    1 [color=red];
    1 -> 2 [color=green];
    1 -> 3;
    5 [color=blue];
    2 -> 3;
    3 -> 5;
    3 -> 4;
    3 -> 6;
}
```

(continues on next page)

(continued from previous page)

```

    4 -> 6;
}

```

```

import pygraphviz as pgv

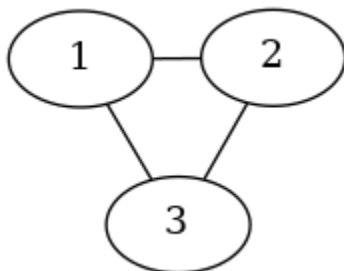
# strict (no parallel edges)
# digraph
# with attribute rankdir set to 'LR'
A = pgv.AGraph(directed=True, strict=True, rankdir="LR")
# add node 1 with color red
A.add_node(1, color="red")
A.add_node(5, color="blue")
# add some edges
A.add_edge(1, 2, color="green")
A.add_edge(2, 3)
A.add_edge(1, 3)
A.add_edge(3, 4)
A.add_edge(3, 5)
A.add_edge(3, 6)
A.add_edge(4, 6)
# adjust a graph parameter
A.graph_attr["epsilon"] = "0.001"
print(A.string()) # print dot file to standard output
A.layout("dot") # layout with dot
A.draw("foo.png") # write to file

```

Total running time of the script: (0 minutes 0.051 seconds)

3.3 Basic

A simple example to create a graphviz dot file and draw a graph.



Out:

```

strict graph "" {
    1 -- 2;
}

```

(continues on next page)

(continued from previous page)

```
1 -- 3;
2 -- 3;
}
```

```
# Copyright (C) 2006 by
# Aric Hagberg <hagberg@lanl.gov>
# Dan Schult <dschult@colgate.edu>
# Manos Renieris, http://www.cs.brown.edu/~er/
# Distributed with BSD license.
# All rights reserved, see LICENSE for details.

__author__ = """Aric Hagberg (hagberg@lanl.gov)"""

import pygraphviz as pgv

A = pgv.AGraph()

A.add_edge(1, 2)
A.add_edge(2, 3)
A.add_edge(1, 3)

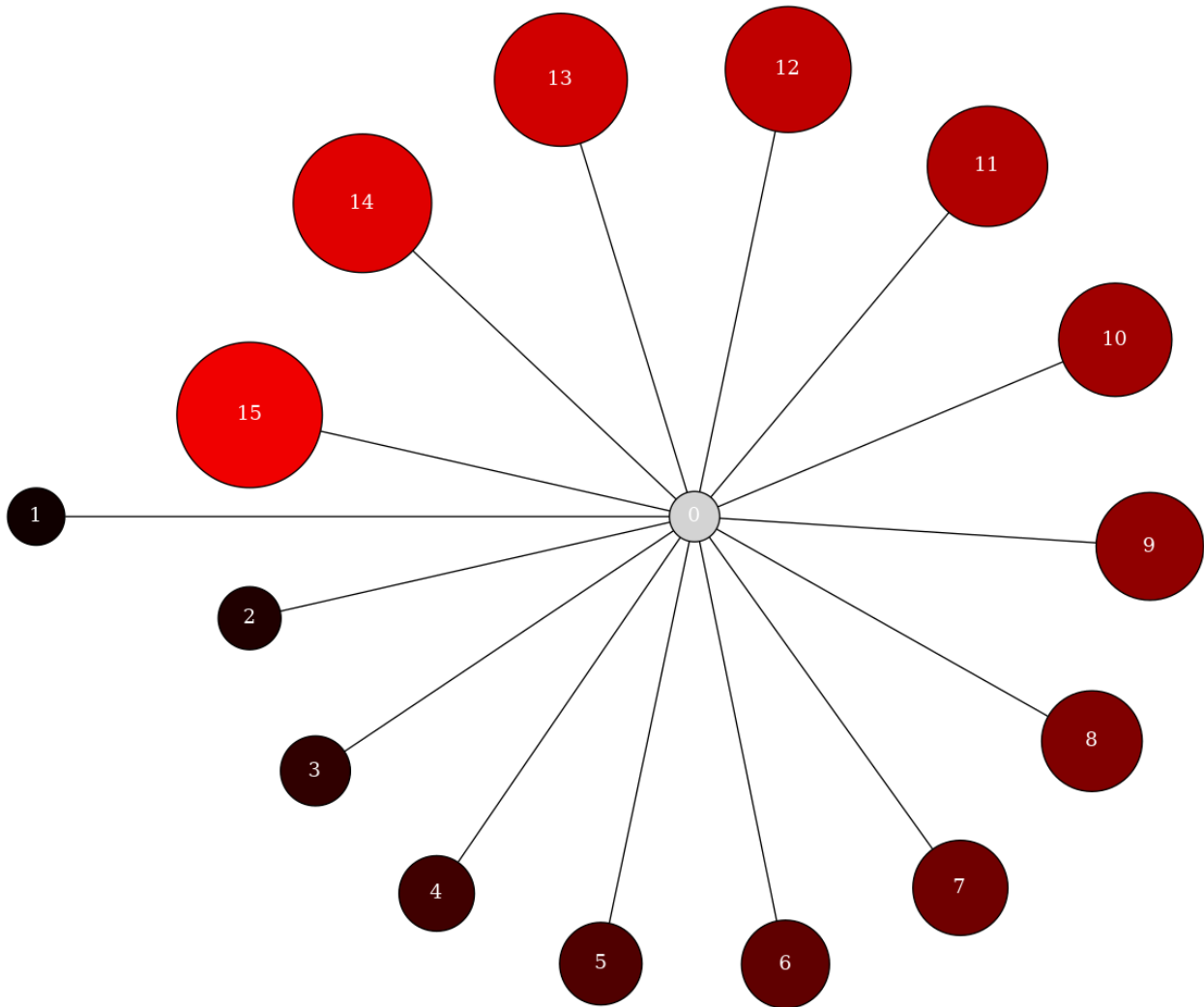
print(A.string()) # print to screen
A.write("simple.dot") # write to simple.dot

B = pgv.AGraph("simple.dot") # create a new graph from file
B.layout() # layout with default (neato)
B.draw("simple.png") # draw png
```

Total running time of the script: (0 minutes 0.045 seconds)

3.4 Star

Create and draw a star with varying node properties.



Out:

```

strict graph "" {
    node [fixedsize=true,
        fontcolor="#FFFFFF",
        shape=circle,
        style=filled
    ];
    1 [fillcolor="#100000",
        height=0.5625,
        width=0.5625];
    0 -- 1;
    2 [fillcolor="#200000",
        height=0.625,
        width=0.625];
    0 -- 2;
    3 [fillcolor="#300000",
        height=0.6875,
        width=0.6875];
    0 -- 3;

```

(continues on next page)

(continued from previous page)

```
4      [fillcolor="#400000",
      height=0.75,
      width=0.75];
0 -- 4;
5      [fillcolor="#500000",
      height=0.8125,
      width=0.8125];
0 -- 5;
6      [fillcolor="#600000",
      height=0.875,
      width=0.875];
0 -- 6;
7      [fillcolor="#700000",
      height=0.9375,
      width=0.9375];
0 -- 7;
8      [fillcolor="#800000",
      height=1.0,
      width=1.0];
0 -- 8;
9      [fillcolor="#900000",
      height=1.0625,
      width=1.0625];
0 -- 9;
10     [fillcolor="#a00000",
      height=1.125,
      width=1.125];
0 -- 10;
11     [fillcolor="#b00000",
      height=1.1875,
      width=1.1875];
0 -- 11;
12     [fillcolor="#c00000",
      height=1.25,
      width=1.25];
0 -- 12;
13     [fillcolor="#d00000",
      height=1.3125,
      width=1.3125];
0 -- 13;
14     [fillcolor="#e00000",
      height=1.375,
      width=1.375];
0 -- 14;
15     [fillcolor="#f00000",
      height=1.4375,
      width=1.4375];
0 -- 15;
}
```

```

# Copyright (C) 2006 by
# Aric Hagberg <hagberg@lanl.gov>
# Dan Schult <dschult@colgate.edu>
# Manos Renieris, http://www.cs.brown.edu/~er/
# Distributed with BSD license.
# All rights reserved, see LICENSE for details.

__author__ = """Aric Hagberg (hagberg@lanl.gov)"""

from pygraphviz import *

A = AGraph()

# set some default node attributes
A.node_attr["style"] = "filled"
A.node_attr["shape"] = "circle"
A.node_attr["fixedsize"] = "true"
A.node_attr["fontcolor"] = "#FFFFFF"

# make a star in shades of red
for i in range(1, 16):
    A.add_edge(0, i)
    n = A.get_node(i)
    n.attr["fillcolor"] = f"#{i * 16:2x}0000"
    n.attr["height"] = f"{i / 16.0 + 0.5}"
    n.attr["width"] = f"{i / 16.0 + 0.5}"

print(A.string()) # print to screen
A.write("star.dot") # write to simple.dot
A.draw("star.png", prog="circo") # draw to png using circo layout

```

Total running time of the script: (0 minutes 0.130 seconds)

3.5 Knuth Miles

An example that shows how to add your own positions to nodes and have graphviz “neato” position the edges.

`miles_graph()` returns an undirected graph over the 128 US cities from the datafile `miles_dat.txt`.

This example is described in Section 1.1 in Knuth’s book¹².

The data used in this example is copied from². The filename and header have been modified to adhere to the request of the author to not corrupt the original source file content and name.

¹ Donald E. Knuth, “The Stanford GraphBase: A Platform for Combinatorial Computing”, ACM Press, New York, 1993.

² <http://www-cs-faculty.stanford.edu/~knuth/sgb.html>

3.5.1 References.



Out:

```
Loaded miles_dat.txt containing 128 cities.  
Wrote miles.dot  
Wrote miles.png
```

```

__author__ = """Aric Hagberg (aric.hagberg@gmail.com)"""

def miles_graph():
    """Return a graph from the data in miles_dat.txt.

    Edges are made between cities that are less than 300 miles apart.
    """
    import math
    import re
    import gzip

    G = pgv.AGraph(name="miles_dat")
    G.node_attr["shape"] = "circle"
    G.node_attr["fixedsize"] = "true"
    G.node_attr["fontsize"] = "8"
    G.node_attr["style"] = "filled"
    G.graph_attr["outputorder"] = "edgesfirst"
    G.graph_attr["label"] = "miles_dat"
    G.graph_attr["ratio"] = "1.0"
    G.edge_attr["color"] = "#1100FF"
    G.edge_attr["style"] = "setlinewidth(2)"

    cities = []
    for line in gzip.open("miles_dat.txt.gz", "rt"):
        if line.startswith("#"): # skip comments
            continue
        numfind = re.compile(r"^\d+")

        if numfind.match(line): # this line is distances
            dist = line.split()
            for d in dist:
                if float(d) < 300: # connect if closer than 300 miles
                    G.add_edge(city, cities[i])
                    i = i + 1
        else: # this line is a city, position, population
            i = 1
            (city, coordpop) = line.split("(")
            cities.insert(0, city)
            (coord, pop) = coordpop.split("]")
            (y, x) = coord.split(",")
            G.add_node(city)
            n = G.get_node(city)
            # assign positions, scale to be something reasonable in points
            n.attr[
                "pos"
            ] = f"{{-(float(x) - 7000) / 10.0:f}},{{(float(y) - 2000) / 10.0:f}}"
            # assign node size, in sqrt of 1,000,000's of people
            d = math.sqrt(float(pop) / 1000000.0)
            n.attr["height"] = f"{{d / 2}}"
            n.attr["width"] = f"{{d / 2}}"
            # assign node color

```

(continues on next page)

(continued from previous page)

```
n.attr["fillcolor"] = f"#0000{int(d * 256):2x}"
# empty labels
n.attr["label"] = " "

return G

if __name__ == "__main__":
    import warnings
    import pygraphviz as pgv

    # ignore Graphviz warning messages
    warnings.simplefilter("ignore", RuntimeWarning)

    G = miles_graph()
    print("Loaded miles_dat.txt containing 128 cities.")

    G.write("miles.dot")
    print("Wrote miles.dot")
    G.draw("miles.png", prog="neato", args="-n2")
    print("Wrote miles.png")
```

Total running time of the script: (0 minutes 0.122 seconds)

REFERENCE

4.1 AGraph Class

class AGraph (*thing=None, filename=None, data=None, string=None, handle=None, name='', strict=True, directed=False, **attr*)

Class for Graphviz agraph type.

Example use

```
>>> import pygraphviz as pgv
>>> G = pgv.AGraph()
>>> G = pgv.AGraph(directed=True)
>>> G = pgv.AGraph("file.dot")
```

Graphviz graph keyword parameters are processed so you may add them like

```
>>> G = pgv.AGraph(landscape="true", ranksep="0.1")
```

or alternatively

```
>>> G = pgv.AGraph()
>>> G.graph_attr.update(landscape="true", ranksep="0.1")
```

and

```
>>> G.node_attr.update(color="red")
>>> G.edge_attr.update(len="2.0", color="blue")
```

See <http://www.graphviz.org/doc/info/attrs.html> for a list of attributes.

Keyword parameters:

thing is a generic input type (filename, string, handle to pointer, dictionary of dictionaries). An attempt is made to automatically detect the type so you may write for example:

```
>>> d = {"1": {"2": None}, "2": {"1": None, "3": None}, "3": {"2": None}}
>>> A = pgv.AGraph(d)
>>> s = A.to_string()
>>> B = pgv.AGraph(s)
>>> h = B.handle
>>> C = pgv.AGraph(h)
```

Parameters:

```

name:      Name for the graph

strict: True|False (True for simple graphs)

directed: True|False

data: Dictionary of dictionaries or dictionary of lists
representing nodes or edges to load into initial graph

string:   String containing a dot format graph

handle:   Swig pointer to an agraph_t data structure

```

Attributes

directed

Return True if graph is directed or False if not.

name

strict

Return True if graph is strict or False if not.

Methods

<i>acyclic</i> ([args, copy])	Reverse sufficient edges in digraph to make graph acyclic.
<i>add_cycle</i> (nlist)	Add the cycle of nodes given in nlist.
<i>add_edge</i> (u[, v, key])	Add a single edge between nodes u and v.
<i>add_edges_from</i> (ebunch, **attr)	Add nodes to graph from a container ebunch.
<i>add_node</i> (n, **attr)	Add a single node n.
<i>add_nodes_from</i> (nbunch, **attr)	Add nodes from a container nbunch.
<i>add_path</i> (nlist)	Add the path of nodes given in nlist.
<i>add_subgraph</i> ([nbunch, name])	Return subgraph induced by nodes in nbunch.
<i>clear</i> ()	Remove all nodes, edges, and attributes from the graph.
<i>copy</i> ()	Return a copy of the graph.
<i>degree</i> ([nbunch, with_labels])	Return the degree of nodes given in nbunch container.
<i>degree_iter</i> ([nbunch, indeg, outdeg])	Return an iterator over the degree of the nodes given in nbunch container.
<i>delete_edge</i> (u[, v, key])	Remove edge between nodes u and v from the graph.
<i>delete_edges_from</i> (ebunch)	Remove edges from ebunch (a container of edges).
<i>delete_node</i> (n)	Remove the single node n.
<i>delete_nodes_from</i> (nbunch)	Remove nodes from a container nbunch.
<i>delete_subgraph</i> (name)	Remove subgraph with given name.
<i>draw</i> ([path, format, prog, args])	Output graph to path in specified format.
<i>edges</i> ([nbunch, keys])	Return list of edges in the graph.
<i>edges_iter</i> ([nbunch, keys])	Return iterator over edges in the graph.
<i>from_string</i> (string)	Load a graph from a string in dot format.

continues on next page

Table 1 – continued from previous page

<code>get_edge(u, v[, key])</code>	Return an edge object (Edge) corresponding to edge (u,v).
<code>get_node(n)</code>	Return a node object (Node) corresponding to node n.
<code>get_subgraph(name)</code>	Return existing subgraph with specified name or None if it doesn't exist.
<code>has_edge(u[, v, key])</code>	Return True an edge u-v is in the graph or False if not.
<code>has_neighbor(u, v[, key])</code>	Return True if u has an edge to v or False if not.
<code>has_node(n)</code>	Return True if n is in the graph or False if not.
<code>in_degree([nbunch, with_labels])</code>	Return the in-degree of nodes given in nbunch container.
<code>in_degree_iter([nbunch])</code>	Return an iterator over the in-degree of the nodes given in nbunch container.
<code>in_edges([nbunch, keys])</code>	Return list of in edges in the graph.
<code>in_edges_iter([nbunch, keys])</code>	Return iterator over out edges in the graph.
<code>in_neighbors(n)</code>	Return list of predecessor nodes of n.
<code>is_directed()</code>	Return True if graph is directed or False if not.
<code>is_strict()</code>	Return True if graph is strict or False if not.
<code>is_undirected()</code>	Return True if graph is undirected or False if not.
<code>iterdegree([nbunch, indeg, outdeg])</code>	Return an iterator over the degree of the nodes given in nbunch container.
<code>iteredges([nbunch, keys])</code>	Return iterator over edges in the graph.
<code>iterindegree([nbunch])</code>	Return an iterator over the in-degree of the nodes given in nbunch container.
<code>iterinedges([nbunch, keys])</code>	Return iterator over out edges in the graph.
<code>iterneighbors(n)</code>	Return iterator over the nodes attached to n.
<code>iternodes()</code>	Return an iterator over all the nodes in the graph.
<code>iteroutdegree([nbunch])</code>	Return an iterator over the out-degree of the nodes given in nbunch container.
<code>iteroutedges([nbunch, keys])</code>	Return iterator over out edges in the graph.
<code>iterpred(n)</code>	Return iterator over predecessor nodes of n.
<code>itersucc(n)</code>	Return iterator over successor nodes of n.
<code>layout([prog, args])</code>	Assign positions to nodes in graph.
<code>neighbors(n)</code>	Return a list of the nodes attached to n.
<code>neighbors_iter(n)</code>	Return iterator over the nodes attached to n.
<code>nodes()</code>	Return a list of all nodes in the graph.
<code>nodes_iter()</code>	Return an iterator over all the nodes in the graph.
<code>number_of_edges()</code>	Return the number of edges in the graph.
<code>number_of_nodes()</code>	Return the number of nodes in the graph.
<code>order()</code>	Return the number of nodes in the graph.
<code>out_degree([nbunch, with_labels])</code>	Return the out-degree of nodes given in nbunch container.
<code>out_degree_iter([nbunch])</code>	Return an iterator over the out-degree of the nodes given in nbunch container.
<code>out_edges([nbunch, keys])</code>	Return list of out edges in the graph.
<code>out_edges_iter([nbunch, keys])</code>	Return iterator over out edges in the graph.
<code>out_neighbors(n)</code>	Return list of successor nodes of n.

continues on next page

Table 1 – continued from previous page

<code>predecessors(n)</code>	Return list of predecessor nodes of n.
<code>predecessors_iter(n)</code>	Return iterator over predecessor nodes of n.
<code>read(path)</code>	Read graph from dot format file on path.
<code>remove_edge(u[, v, key])</code>	Remove edge between nodes u and v from the graph.
<code>remove_edges_from(ebunch)</code>	Remove edges from ebunch (a container of edges).
<code>remove_node(n)</code>	Remove the single node n.
<code>remove_nodes_from(nbunch)</code>	Remove nodes from a container nbunch.
<code>remove_subgraph(name)</code>	Remove subgraph with given name.
<code>reverse()</code>	Return copy of directed graph with edge directions reversed.
<code>string()</code>	Return a string (unicode) representation of graph in dot format.
<code>string_nop()</code>	Return a string (unicode) representation of graph in dot format.
<code>subgraph([nbunch, name])</code>	Return subgraph induced by nodes in nbunch.
<code>subgraph_parent([nbunch, name])</code>	Return parent graph of subgraph or None if graph is root graph.
<code>subgraph_root([nbunch, name])</code>	Return root graph of subgraph or None if graph is root graph.
<code>subgraphs()</code>	Return a list of all subgraphs in the graph.
<code>subgraphs_iter()</code>	Iterator over subgraphs.
<code>successors(n)</code>	Return list of successor nodes of n.
<code>successors_iter(n)</code>	Return iterator over successor nodes of n.
<code>to_directed(**kwds)</code>	Return directed copy of graph.
<code>to_string()</code>	Return a string representation of graph in dot format.
<code>to_undirected()</code>	Return undirected copy of graph.
<code>tred([args, copy])</code>	Transitive reduction of graph.
<code>unflatten([args])</code>	Adjust directed graphs to improve layout aspect ratio.
<code>write([path])</code>	Write graph in dot format to file on path.

close	
get_name	

acyclic (*args*='', *copy*=False)

Reverse sufficient edges in digraph to make graph acyclic. Modifies existing graph.

To create a new graph use

```
>>> import pygraphviz as pgv
>>> A = pgv.AGraph(directed=True)
>>> B = A.acyclic(copy=True)
```

See the graphviz “acyclic” program for details of the algorithm.

add_cycle (*nlist*)

Add the cycle of nodes given in nlist.

add_edge (*u*, *v=None*, *key=None*, ***attr*)

Add a single edge between nodes *u* and *v*.

If the nodes *u* and *v* are not in the graph they will be added.

If *u* and *v* are not strings, conversion to a string will be attempted. String conversion will work if *u* and *v* have valid string representation (try `str(u)` if you are unsure).

```
>>> import pygraphviz as pgv
>>> G = pgv.AGraph()
>>> G.add_edge("a", "b")
>>> G.edges()
[('a', 'b')]
```

The optional *key* argument allows assignment of a key to the edge. This is especially useful to distinguish between parallel edges in multi-edge graphs (*strict=False*).

```
>>> G = pgv.AGraph(strict=False)
>>> G.add_edge("a", "b", "first")
>>> G.add_edge("a", "b", "second")
>>> sorted(G.edges(keys=True))
[('a', 'b', 'first'), ('a', 'b', 'second')]
```

Attributes can be added when edges are created or updated after creation

```
>>> G.add_edge("a", "b", color="green")
```

Attributes must be valid strings.

See <http://www.graphviz.org/doc/info/attrs.html> for a list of attributes.

add_edges_from (*ebunch*, ***attr*)

Add nodes to graph from a container *ebunch*.

ebunch is a container of edges such as a list or dictionary.

```
>>> import pygraphviz as pgv
>>> G = pgv.AGraph()
>>> elist = [("a", "b"), ("b", "c")]
>>> G.add_edges_from(elist)
```

Attributes can be added when edges are created or updated after creation

```
>>> G.add_edges_from(elist, color="green")
```

add_node (*n*, ***attr*)

Add a single node *n*.

If *n* is not a string, conversion to a string will be attempted. String conversion will work if *n* has valid string representation (try `str(n)` if you are unsure).

```
>>> import pygraphviz as pgv
>>> G = pgv.AGraph()
>>> G.add_node("a")
>>> G.nodes()
['a']
>>> G.add_node(1)  # will be converted to a string
```

(continues on next page)

(continued from previous page)

```
>>> G.nodes()
['a', '1']
```

Attributes can be added to nodes on creation or updated after creation (attribute values must be strings)

```
>>> G.add_node(2, color="red")
```

See <http://www.graphviz.org/doc/info/attrs.html> for a list of attributes.

Anonymous Graphviz nodes are currently not implemented.

add_nodes_from(nbunch, **attr)

Add nodes from a container nbunch.

nbunch can be any iterable container such as a list or dictionary

```
>>> import pygraphviz as pgv
>>> G = pgv.AGraph()
>>> nlist = ["a", "b", 1, "spam"]
>>> G.add_nodes_from(nlist)
>>> sorted(G.nodes())
['1', 'a', 'b', 'spam']
```

Attributes can be added to nodes on creation or updated after creation

```
>>> G.add_nodes_from(nlist, color="red")  # set all nodes in nlist red
```

add_path(nlist)

Add the path of nodes given in nlist.

add_subgraph(nbunch=None, name=None, **attr)

Return subgraph induced by nodes in nbunch.

clear()

Remove all nodes, edges, and attributes from the graph.

close()

copy()

Return a copy of the graph.

Notes

Versions <=1.6 made a copy by writing and the reading a dot string. This version loads a new graph with nodes, edges and attributes.

degree(nbunch=None, with_labels=False)

Return the degree of nodes given in nbunch container.

Using optional with_labels=True returns a dictionary keyed by node with value set to the degree.

degree_iter(nbunch=None, indeg=True, outdeg=True)

Return an iterator over the degree of the nodes given in nbunch container.

Returns pairs of (node,degree).

delete_edge (*u*, *v=None*, *key=None*)

Remove edge between nodes *u* and *v* from the graph.

With optional key argument will only remove an edge matching (*u,v,key*).

delete_edges_from (*ebunch*)

Remove edges from *ebunch* (a container of edges).

delete_node (*n*)

Remove the single node *n*.

Attempting to remove a node that isn't in the graph will produce an error.

```
>>> import pygraphviz as pgv
>>> G = pgv.AGraph()
>>> G.add_node("a")
>>> G.remove_node("a")
```

delete_nodes_from (*nbunch*)

Remove nodes from a container *nbunch*.

nbunch can be any iterable container such as a list or dictionary

```
>>> import pygraphviz as pgv
>>> G = pgv.AGraph()
>>> nlist = ["a", "b", 1, "spam"]
>>> G.add_nodes_from(nlist)
>>> G.remove_nodes_from(nlist)
```

delete_subgraph (*name*)

Remove subgraph with given name.

property directed

Return True if graph is directed or False if not.

draw (*path=None*, *format=None*, *prog=None*, *args=''*)

Output graph to *path* in specified format.

An attempt will be made to guess the output format based on the file extension of *path*. If that fails, then the *format* parameter will be used.

Note, if *path* is a file object returned by a call to `os.fdopen()`, then the method for discovering the format will not work. In such cases, one should explicitly set the *format* parameter; otherwise, it will default to 'dot'.

If *path* is None, the result is returned as a Bytes object.

Formats (not all may be available on every system depending on how Graphviz was built)

```
'canon', 'cmap', 'cmapx', 'cmapx_np', 'dia', 'dot', 'fig', 'gd', 'gd2', 'gif', 'hpgl', 'imap',
'imap_np', 'ismap', 'jpe', 'jpeg', 'jpg', 'mif', 'mp', 'pcl', 'pdf', 'pic', 'plain', 'plain-ext',
'png', 'ps', 'ps2', 'svg', 'svgz', 'vml', 'vmlz', 'vrml', 'vtx', 'wbmp', 'xdot', 'xlib'
```

If *prog* is not specified and the graph has positions (see `layout()`) then no additional graph positioning will be performed.

Optional *prog*=['neato'|'dot'|'twopi'|'circo'|'fdp'|'nop'] will use specified graphviz layout method.

```
>>> import pygraphviz as pgv
>>> G = pgv.AGraph()
>>> G.add_edges_from([(0, 1), (1, 2), (2, 0), (2, 3)])
>>> G.layout()
```

use current node positions, output pdf in 'file.pdf' >>> G.draw("file.pdf")

use dot to position, output png in 'file' >>> G.draw("file", format="png", prog="dot")

use keyword 'args' to pass additional arguments to graphviz >>> G.draw("test.pdf",
prog="twopi", args="-Gepsilon=1") >>> G.draw("test2.pdf", args="-Nshape=box -
Edir=forward -Ecolor=red")

The layout might take a long time on large graphs.

edges (nbunch=None, keys=False)

Return list of edges in the graph.

If the optional nbunch (container of nodes) only edges adjacent to nodes in nbunch will be returned.

```
>>> import pygraphviz as pgv
>>> G = pgv.AGraph()
>>> G.add_edge("a", "b")
>>> G.add_edge("c", "d")
>>> print(sorted(G.edges()))
[('a', 'b'), ('c', 'd')]
>>> print(G.edges("a"))
[('a', 'b')]
```

edges_iter (nbunch=None, keys=False)

Return iterator over edges in the graph.

If the optional nbunch (container of nodes) only edges adjacent to nodes in nbunch will be returned.

Note: modifying the graph structure while iterating over edges may produce unpredictable results. Use edges() as an alternative.

from_string (string)

Load a graph from a string in dot format.

Overwrites any existing graph.

To make a new graph from a string use

```
>>> import pygraphviz as pgv
>>> s = "digraph {1 -> 2}"
>>> A = pgv.AGraph()
>>> t = A.from_string(s)
>>> A = pgv.AGraph(string=s) # specify s is a string
>>> A = pgv.AGraph(s) # s assumed to be a string during_
↪ initialization
```

get_edge (u, v, key=None)

Return an edge object (Edge) corresponding to edge (u,v).

```
>>> import pygraphviz as pgv
>>> G = pgv.AGraph()
>>> G.add_edge("a", "b")
>>> edge = G.get_edge("a", "b")
>>> print(edge)
('a', 'b')
```

With optional key argument will only get edge matching (u,v,key).

get_name()

get_node(n)

Return a node object (Node) corresponding to node n.

```
>>> import pygraphviz as pgv
>>> G = pgv.AGraph()
>>> G.add_node("a")
>>> node = G.get_node("a")
>>> print(node)
a
```

get_subgraph(name)

Return existing subgraph with specified name or None if it doesn't exist.

has_edge(u, v=None, key=None)

Return True an edge u-v is in the graph or False if not.

```
>>> import pygraphviz as pgv
>>> G = pgv.AGraph()
>>> G.add_edge("a", "b")
>>> G.has_edge("a", "b")
True
```

Optional key argument will restrict match to edges (u,v,key).

has_neighbor(u, v, key=None)

Return True if u has an edge to v or False if not.

```
>>> import pygraphviz as pgv
>>> G = pgv.AGraph()
>>> G.add_edge("a", "b")
>>> G.has_neighbor("a", "b")
True
```

Optional key argument will only find edges (u,v,key).

has_node(n)

Return True if n is in the graph or False if not.

```
>>> import pygraphviz as pgv
>>> G = pgv.AGraph()
>>> G.add_node("a")
>>> G.has_node("a")
True
```

(continues on next page)

(continued from previous page)

```
>>> "a" in G # same as G.has_node('a')
True
```

in_degree (*nbunch=None, with_labels=False*)

Return the in-degree of nodes given in nbunch container.

Using optional with_labels=True returns a dictionary keyed by node with value set to the degree.

in_degree_iter (*nbunch=None*)

Return an iterator over the in-degree of the nodes given in nbunch container.

Returns pairs of (node,degree).

in_edges (*nbunch=None, keys=False*)

Return list of in edges in the graph. If the optional nbunch (container of nodes) only in edges adjacent to nodes in nbunch will be returned.

in_edges_iter (*nbunch=None, keys=False*)

Return iterator over out edges in the graph.

If the optional nbunch (container of nodes) only out edges adjacent to nodes in nbunch will be returned.

Note: modifying the graph structure while iterating over edges may produce unpredictable results. Use in_edges() as an alternative.

in_neighbors (*n*)

Return list of predecessor nodes of n.

is_directed ()

Return True if graph is directed or False if not.

is_strict ()

Return True if graph is strict or False if not.

Strict graphs do not allow parallel edges or self loops.

is_undirected ()

Return True if graph is undirected or False if not.

iterdegree (*nbunch=None, indeg=True, outdeg=True*)

Return an iterator over the degree of the nodes given in nbunch container.

Returns pairs of (node,degree).

iteredges (*nbunch=None, keys=False*)

Return iterator over edges in the graph.

If the optional nbunch (container of nodes) only edges adjacent to nodes in nbunch will be returned.

Note: modifying the graph structure while iterating over edges may produce unpredictable results. Use edges() as an alternative.

iterindegree (*nbunch=None*)

Return an iterator over the in-degree of the nodes given in nbunch container.

Returns pairs of (node,degree).

iterinedges (*nbunch=None, keys=False*)

Return iterator over out edges in the graph.

If the optional nbunch (container of nodes) only out edges adjacent to nodes in nbunch will be returned.

Note: modifying the graph structure while iterating over edges may produce unpredictable results. Use `in_edges()` as an alternative.

iterneighbors (*n*)

Return iterator over the nodes attached to *n*.

Note: modifying the graph structure while iterating over node neighbors may produce unpredictable results. Use `neighbors()` as an alternative.

iternodes ()

Return an iterator over all the nodes in the graph.

Note: modifying the graph structure while iterating over the nodes may produce unpredictable results. Use `nodes()` as an alternative.

iteroutdegree (*nbunch=None*)

Return an iterator over the out-degree of the nodes given in nbunch container.

Returns pairs of (node,degree).

iteroutedges (*nbunch=None, keys=False*)

Return iterator over out edges in the graph.

If the optional nbunch (container of nodes) only out edges adjacent to nodes in nbunch will be returned.

Note: modifying the graph structure while iterating over edges may produce unpredictable results. Use `out_edges()` as an alternative.

iterpred (*n*)

Return iterator over predecessor nodes of *n*.

Note: modifying the graph structure while iterating over node predecessors may produce unpredictable results. Use `predecessors()` as an alternative.

itersucc (*n*)

Return iterator over successor nodes of *n*.

Note: modifying the graph structure while iterating over node successors may produce unpredictable results. Use `successors()` as an alternative.

layout (*prog='neato', args=''*)

Assign positions to nodes in graph.

Optional `prog=['neato'|'dot'|'twopi'|'circo'|'fdp'|'nop']` will use specified graphviz layout method.

```
>>> import pygraphviz as pgv
>>> A = pgv.AGraph()
>>> A.add_edge(1, 2)
>>> A.layout()
>>> A.layout(prog="neato", args="-Nshape=box -Efontsize=8")
```

Use keyword args to add additional arguments to graphviz programs.

The layout might take a long time on large graphs.

property name

neighbors (*n*)

Return a list of the nodes attached to *n*.

neighbors_iter (*n*)

Return iterator over the nodes attached to *n*.

Note: modifying the graph structure while iterating over node neighbors may produce unpredictable results. Use `neighbors()` as an alternative.

nodes ()

Return a list of all nodes in the graph.

nodes_iter ()

Return an iterator over all the nodes in the graph.

Note: modifying the graph structure while iterating over the nodes may produce unpredictable results. Use `nodes()` as an alternative.

number_of_edges ()

Return the number of edges in the graph.

number_of_nodes ()

Return the number of nodes in the graph.

order ()

Return the number of nodes in the graph.

out_degree (*nbunch=None, with_labels=False*)

Return the out-degree of nodes given in *nbunch* container.

Using optional *with_labels=True* returns a dictionary keyed by node with value set to the degree.

out_degree_iter (*nbunch=None*)

Return an iterator over the out-degree of the nodes given in *nbunch* container.

Returns pairs of (node,degree).

out_edges (*nbunch=None, keys=False*)

Return list of out edges in the graph.

If the optional *nbunch* (container of nodes) only out edges adjacent to nodes in *nbunch* will be returned.

out_edges_iter (*nbunch=None, keys=False*)

Return iterator over out edges in the graph.

If the optional *nbunch* (container of nodes) only out edges adjacent to nodes in *nbunch* will be returned.

Note: modifying the graph structure while iterating over edges may produce unpredictable results. Use `out_edges()` as an alternative.

out_neighbors (*n*)

Return list of successor nodes of *n*.

predecessors (*n*)

Return list of predecessor nodes of *n*.

predecessors_iter (*n*)

Return iterator over predecessor nodes of *n*.

Note: modifying the graph structure while iterating over node predecessors may produce unpredictable results. Use `predecessors()` as an alternative.

read (*path*)

Read graph from dot format file on *path*.

path can be a file name or file handle

use:

```
G.read('file.dot')
```

remove_edge (*u, v=None, key=None*)

Remove edge between nodes *u* and *v* from the graph.

With optional *key* argument will only remove an edge matching (*u,v,key*).

remove_edges_from (*ebunch*)

Remove edges from *ebunch* (a container of edges).

remove_node (*n*)

Remove the single node *n*.

Attempting to remove a node that isn't in the graph will produce an error.

```
>>> import pygraphviz as pgv
>>> G = pgv.AGraph()
>>> G.add_node("a")
>>> G.remove_node("a")
```

remove_nodes_from (*nbunch*)

Remove nodes from a container *nbunch*.

nbunch can be any iterable container such as a list or dictionary

```
>>> import pygraphviz as pgv
>>> G = pgv.AGraph()
>>> nlist = ["a", "b", 1, "spam"]
>>> G.add_nodes_from(nlist)
>>> G.remove_nodes_from(nlist)
```

remove_subgraph (*name*)

Remove subgraph with given *name*.

reverse ()

Return copy of directed graph with edge directions reversed.

property strict

Return True if graph is strict or False if not.

Strict graphs do not allow parallel edges or self loops.

string ()

Return a string (unicode) representation of graph in dot format.

string_nop()

Return a string (unicode) representation of graph in dot format.

subgraph (*nbunch=None, name=None, **attr*)

Return subgraph induced by nodes in nbunch.

subgraph_parent (*nbunch=None, name=None*)

Return parent graph of subgraph or None if graph is root graph.

subgraph_root (*nbunch=None, name=None*)

Return root graph of subgraph or None if graph is root graph.

subgraphs()

Return a list of all subgraphs in the graph.

subgraphs_iter()

Iterator over subgraphs.

successors (*n*)

Return list of successor nodes of n.

successors_iter (*n*)

Return iterator over successor nodes of n.

Note: modifying the graph structure while iterating over node successors may produce unpredictable results. Use `successors()` as an alternative.

to_directed (***kws*)

Return directed copy of graph.

Each undirected edge u-v is represented as two directed edges u->v and v->u.

to_string()

Return a string representation of graph in dot format.

`to_string()` uses “agwrite” to produce “dot” format w/o rendering. The function `string_nop()` layouts with “nop” and renders to “dot”.

to_undirected()

Return undirected copy of graph.

tred (*args='', copy=False*)

Transitive reduction of graph. Modifies existing graph.

To create a new graph use

```
>>> import pygraphviz as pgv
>>> A = pgv.AGraph(directed=True)
>>> B = A.tred(copy=True)
```

See the graphviz “tred” program for details of the algorithm.

unflatten (*args=''*)

Adjust directed graphs to improve layout aspect ratio.

```
>>> import pygraphviz as pgv
>>> A = pgv.AGraph()
>>> A_unflattened = A.unflatten("-f -l 3")
>>> A.unflatten("-f -l 1").layout()
```

Use keyword args to add additional arguments to graphviz programs.

`write` (*path=None*)

Write graph in dot format to file on path.

path can be a file name or file handle

use:

```
G.write('file.dot')
```

4.2 FAQ

Q

I followed the installation instructions but when I do:

```
>>> import pygraphviz
```

I get an error like:

```
ImportError: libagraph.so.1: cannot open shared object
file: No such file or directory
```

What is wrong?

A

Some Unix systems don't include the Graphviz library in the default search path for the run-time linker. The path is often something like `/usr/lib/graphviz` or `/sw/lib/graphviz` etc. and it needs to be added to your search path. On *nix systems, the preferred way to do this is by setting the appropriate flags when building/installing `pygraphviz`. For example, if the Graphviz libraries are installed in `/opt/lib/mygviz/` on your system:

```
pip install --global-option=build_ext \
            --global-option="-L/opt/lib/mygviz/" \
            --global-option="-R/opt/lib/mygviz/" \
            pygraphviz
```

In this example, the `-L` and `-R` flags tell the linker where to look for the required Graphviz libraries at build time and run time, respectively.

Q

How do I compile `pygraphviz` under Windows?

A

See [Windows](#) for the latest on how to install Graphviz and `pygraphviz` on Windows.

Q

Why don't you distribute a `pygraphviz` Windows installer?

A

We would very much like to make binary wheels available for `pygraphviz`, but there are several complications. `pygraphviz` is a wrapper around Graphviz, which means that Graphviz must be installed, and Graphviz header files, libraries *and* command line executables must all be accessible for the wrapper. The recommended use of the [Graphviz CLI](#) poses challenges for wheel packaging.

See also:

This [GitHub issue](#) for further discussion on wheels and packaging.

4.3 API Notes

4.3.1 pygraphviz-1.2

No API changes

4.3.2 pygraphviz-1.1

Pygraphviz-1.1 adds unicode (graphviz charset) support. The default Node type is now unicode. See `examples/utf8.py` for an example of how to use non-ASCII characters.

The `__str__` and `__repr__` methods have been rewritten and a `__unicode__` method added.

If `G` is a `pygraphviz.AGraph` object then

- `str(G)` produces a dot-format string representation (some characters might not be represented correctly)
- `unicode(G)` produces a dot-format unicode representation
- `repr(G)` produces a string of the unicode representation.
- `print G` produces a formatted dot language output

4.3.3 pygraphviz-0.32

pygraphviz-0.32 is a rewrite of pygraphviz-0.2x with some significant changes in the API and Graphviz wrapper. It is not compatible with earlier versions.

The goal of pygraphviz is to provide a (mostly) Pythonic interface to the Graphviz Agraph data-structure, layout, and drawing algorithms.

The API is now similar to the NetworkX API. Studying the documentation and Tutorial for NetworkX will teach you most of what you need to know for pygraphviz. For a short introduction on pygraphviz see the pygraphviz Tutorial.

There are some important differences between the PyGraphviz and NetworkX API. With PyGraphviz

- All nodes must be of string or unicode type. An attempt will be made to convert other types to a string.
- Nodes and edges are custom Python objects. Nodes are like unicode/string objects and edges are like tuple objects. (In NetworkX nodes can be anything and edges are two- or three-tuples.)
- Graphs, edges, and nodes may have attributes such as color, size, shape, attached to them. If the attributes are known Graphviz attributes they will be used for drawing and layout.
- The `layout()` and `draw()` methods allow positioning of nodes and rendering in all of the supported Graphviz output formats.
- The `string()` method produces a string with the graph represented in Graphviz dot format. See also `from_string()`.

- The `subgraph()` method is the Graphviz representation of subgraphs: a tree of graphs under the original (root) graph. They are primarily used for clustering of nodes when drawing with `dot`.

Pygraphviz supports most of the Graphviz API.

4.4 News

4.4.1 pygraphviz-1.9

Release date: 9 February 2022

- Drop Python 3.7 support
- Add Python 3.10 support
- Add `osage` and `patchwork` to `progs` list
- Add IPython rich display hook to `AGraph` class
- Add contributor guide
- Fixed directed nature of `AGraph.copy()`
- Minor documentation and code fixes

4.4.2 pygraphviz-1.8

Release date: 20 January 2022

This release was pulled because the install was broken with `pip 22` and `python 3.7`.

4.4.3 pygraphviz-1.7

Release date: 1 February 2021

- Drop Python 3.6 support
- Add Python 3.9 support
- Require Graphviz 2.42+, (Graphviz 2.46+ recommended)
- Improve installation process and documentation
- Switch from `nose` to `pytest`
- Remove old Python 2 code
- `AGraph.eq` includes attribute comparison (PR #246)

4.4.4 pygraphviz-1.6

Release date: 05 September 2020

- Add Python 3.8 support
- Drop Python 2.7 support
- Update to SWIG 4.0.1

4.4.5 pygraphviz-1.5

Release date: 10 September 2018

- Python 3.7 support

4.4.6 pygraphviz-1.3.1

Release date: 6 September 2015

- Update manifest to include missing files

4.4.7 pygraphviz-1.3

Release date: 5 September 2015

- Python 3 support
- Encoding bugfixes

<https://github.com/pygraphviz/pygraphviz/issues?q=milestone%3Apygraphviz-1.3+is%3Aclosed>

4.4.8 pygraphviz-1.2

Release date: 3 August 2013

- Quote Graphviz program names to work with space (Windows fix)
- Keep name in reverse()

4.4.9 pygraphviz-1.1

Release date: 9 February 2011

- Added unicode support for handling non-ASCII characters
- Better handling of user data on initialization of AGraph() object to guess input type (AGraph object, file, dict-of-dicts, file)
- Add sfdp to layout options

See <https://networkx.lanl.gov/trac/query?group=status&milestone=pygraphviz-1.1>

4.4.10 pygraphviz-1.0.0

Release date: 30 July 2010

See: <https://networkx.lanl.gov/trac/timeline>

- Added to_string() and from_string methods
- Interface to graphviz “acyclic” and “tred”
- Better handling of user data on initialization of AGraph() object to guess input type (AGraph object, file, dict-of-dicts, file)
- Add handling of default attributes for subgraphs
- Improved error handling when using non-string data
- Fix bug in default attribute handling
- Make sure file handles are closed correctly

4.4.11 pygraphviz-0.99.1

Release date: 7 December 2008

See: <https://networkx.lanl.gov/trac/timeline>

- Use Graphviz libgraph instead of deprecated libagraph
- More closely match API to NetworkX
- edges() now produces two-tuples or three tuples if edges(keys=True)
- Edge and Node objects now have .name and .handle properties
- Warn without throwing exceptions for Graphviz errors
- Graph now has .strict and .directed properties
- Cleared up fontsize warnings in examples

4.4.12 pygraphviz-0.99

Release date: 18 November 2008

See: <https://networkx.lanl.gov/trac/timeline>

- New documentation at <http://networkx.lanl.gov/pygraphviz/>
- Developer’s site at <https://networkx.lanl.gov/trac/wiki/PyGraphviz>

4.4.13 pygraphviz-0.37

Release date: 17 August 2008

See: <https://networkx.lanl.gov/trac/timeline>

- Handle default attributes for subgraphs, examples at <https://networkx.lanl.gov/trac/browser/pygraphviz/trunk/doc/examples/attributes.py> <https://networkx.lanl.gov/trac/browser/pygraphviz/trunk/doc/examples/subgraph.py>
- Buggy attribute assignment fixed by Graphviz team (use Graphviz>2.17.20080127)

- Encode all strings as UTF-8 as default
- Fix AGraph.clear() memory leak and attempt to address slow deletion of nodes and edges
- Allow pdf output and support all available output types on a given platform
- Fix number_of_edges() to use gv.agnedges to correctly report edges for graphs with self loops

4.4.14 pygraphviz-0.36

Release date: 13 January 2008

See: <https://networkx.lanl.gov/trac/timeline>

- Automatic handling of types on init of AGraph(data): data can be a filename, string in dot format, dictionary-of-dictionaries, or a SWIG AGraph pointer.
- Add interface to Graphviz programs acyclic and tred
- Refactor process handling to allow easier access to Graphviz layout and graph processing programs
- to_string() and from_string() methods
- Handle multiple anonymous edges correctly
- Attribute handling on add_node, add_edge and init of AGraph. So you can e.g. `A=AGraph(ranksep='0.1'); A.add_node('a',color='red') A.add_edge('a','b',color='blue')`

4.4.15 pygraphviz-0.35

Release date: 22 July 2007

See: <https://networkx.lanl.gov/trac/timeline>

- Rebuilt SWIG wrappers - works correctly now on 64 bit machines/python2.5
- Implement Graphviz subgraph functionality
- Better error reporting when attempting to set attributes, avoid segfault when using None
- pkg-config handling now works in more configurations (hopefully all)

4.4.16 pygraphviz-0.34

Release date: 11 April 2007

See: <https://networkx.lanl.gov/trac/timeline>

- run “python setup_egg.py test” for tests if you have setuptools
- added tests for layout code
- use pkg-config for finding graphviz (dotneato-config still works for older graphviz versions)
- use threads and temporary files for multiplatform nonblocking IO
- django example

4.4.17 pygraphviz-0.33

- Workaround for “nop” bug in graphviz-2.8, improved packaging, updated swig wrapper, better error handling.

4.4.18 pygraphviz-0.32

The release pygraphviz-0.32 is the second rewrite of the original project. It has improved attribute handling and drawing capabilities. It is not backward compatible with earlier versions. Earlier versions will always be available at the download site.

This version now inter-operates with many of the NetworkX algorithms and graph generators. See https://networkx.lanl.gov/trac/browser/networkx/trunk/doc/examples/pygraphviz_simple.py

4.5 Related Packages

- Python bindings distributed with Graphviz (graphviz-python): <http://www.graphviz.org/>
- pydot: <http://code.google.com/p/pydot/>
- mfGraph: <http://www.geocities.com/foetsch/mfgraph/index.htm>
- Yapgvb: <http://yapgvb.sourceforge.net/>

4.6 History

The original concept was developed and implemented by Manos Renieris at Brown University: <http://www.cs.brown.edu/~er/software/>

4.7 Credits

Thanks to Stephen North and the AT&T Graphviz team for creating and maintaining the Graphviz graph layout and drawing packages

Thanks to Manos Renieris for the original idea.

Thanks to the following people who have made contributions:

- Cyril Brulebois helped clean up the packaging for Debian and find bugs.
- Rene Hogendoorn developed the threads code to provide nonblocking, multiplatform IO.
- Ross Richardson suggested fixes and tested the attribute handling.
- Alexis Dinno debugged the setup and installation for OSX.
- Stefano Costa reported attribute bugs and contributed the code to run Graphviz “tred” and friends.
- Casey Deccio contributed unicode handling design and code.

4.8 Legal

4.8.1 PyGraphviz License

Copyright (C) 2004-2010 by Aric Hagberg <hagberg@lanl.gov> Dan Schult <dschult@colgate.edu> Manos Renieris, <http://www.cs.brown.edu/~er/> Distributed with BSD license. All rights reserved, see LICENSE for details.

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

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the <ORGANIZATION> nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

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

4.8.2 Notice

This software and ancillary information (herein called SOFTWARE) called pygraphviz is made available under the terms described here. The SOFTWARE has been approved for release with associated LA-CC number 04-073.

Unless otherwise indicated, this SOFTWARE has been authored by an employee or employees of the University of California, operator of the Los Alamos National Laboratory under Contract No. W-7405-ENG-36 with the U.S. Department of Energy. The U.S. Government has rights to use, reproduce, and distribute this SOFTWARE. The public may copy, distribute, prepare derivative works and publicly display this SOFTWARE without charge, provided that this Notice and any statement of authorship are reproduced on all copies. Neither the Government nor the University makes any warranty, express or implied, or assumes any liability or responsibility for the use of this SOFTWARE.

If SOFTWARE is modified to produce derivative works, such modified SOFTWARE should be clearly marked, so as not to confuse it with the version available from Los Alamos National Laboratory.

CONTRIBUTOR GUIDE

Note: This document assumes some familiarity with contributing to open source scientific Python projects using GitHub pull requests.

5.1 Development Workflow

1. If you are a first-time contributor:

- You'll need to install GraphViz on your local computer, if you haven't already. For OS-specific installation instructions, refer to [INSTALL.txt](#).
- Go to <https://github.com/pygraphviz/pygraphviz> and click the “fork” button to create your own copy of the project.
- Clone the project to your local computer:

```
git clone git@github.com:your-username/pygraphviz.git
```

- Navigate to the folder pygraphviz and add the upstream repository:

```
git remote add upstream git@github.com:pygraphviz/pygraphviz.git
```

- Now, you have remote repositories named:
 - upstream, which refers to the pygraphviz repository
 - origin, which refers to your personal fork
- Next, you need to set up your build environment. Here are instructions using venv:
 - venv (pip based)

```
# Create a virtualenv named ``pygraphviz-dev`` that lives in the
↳directory of
# the same name
python -m venv pygraphviz-dev
# Activate it
source pygraphviz-dev/bin/activate
# Install main development and runtime dependencies of pygraphviz
pip install -r requirements/test.txt -r requirements/developer.txt
#
```

(continues on next page)

(continued from previous page)

```
# Build and install pygraphviz from source
pip install -e .
# Test your installation
PYTHONPATH=. pytest pygraphviz
```

- Finally, we recommend you use a pre-commit hook, which runs black when you type git commit:

```
pre-commit install
```

2. Develop your contribution:

- Pull the latest changes from upstream:

```
git checkout main
git pull upstream main
```

- Create a branch for the feature you want to work on. Since the branch name will appear in the merge message, use a sensible name such as ‘bugfix-for-issue-1480’:

```
git checkout -b bugfix-for-issue-1480
```

- Commit locally as you progress (git add and git commit)

3. Test your contribution:

- Run the test suite locally (see [Testing](#) for details):

```
PYTHONPATH=. pytest pygraphviz
```

- Running the tests locally *before* submitting a pull request helps catch problems early and reduces the load on the continuous integration system.

4. Submit your contribution:

- Push your changes back to your fork on GitHub:

```
git push origin bugfix-for-issue-1480
```

- Go to GitHub. The new branch will show up with a green Pull Request button—click it.
- If you want, post on the [mailing list](#) to explain your changes or to ask for review.

5. Review process:

- Every Pull Request (PR) update triggers a set of [continuous integration](#) services that check that the code is up to standards and passes all our tests. These checks must pass before your PR can be merged. If one of the checks fails, you can find out why by clicking on the “failed” icon (red cross) and inspecting the build and test log.
- Reviewers (the other developers and interested community members) will write inline and/or general comments on your PR to help you improve its implementation, documentation, and style. Every single developer working on the project has their code reviewed, and we’ve come to see it as friendly conversation from which we all learn and the overall code quality benefits. Therefore, please don’t let the review discourage you from contributing: its only aim is to improve the quality of project, not to criticize (we are, after all, very grateful for the time you’re donating!).

- To update your PR, make your changes on your local repository and commit. As soon as those changes are pushed up (to the same branch as before) the PR will update automatically.

Note: If the PR closes an issue, make sure that GitHub knows to automatically close the issue when the PR is merged. For example, if the PR closes issue number 1480, you could use the phrase “Fixes #1480” in the PR description or commit message.

To reviewers: make sure the merge message also has a brief description of the change(s).

5.2 Divergence from upstream main

If GitHub indicates that the branch of your Pull Request can no longer be merged automatically, merge the main branch into yours:

```
git fetch upstream main
git merge upstream/main
```

If any conflicts occur, they need to be fixed before continuing. See which files are in conflict using:

```
git status
```

Which displays a message like:

```
Unmerged paths:
  (use "git add <file>..." to mark resolution)

   both modified:   file_with_conflict.txt
```

Inside the conflicted file, you'll find sections like these:

```
<<<<<< HEAD
The way the text looks in your branch
=====
The way the text looks in the main branch
>>>>>> main
```

Choose one version of the text that should be kept, and delete the rest:

```
The way the text looks in your branch
```

Now, add the fixed file:

```
git add file_with_conflict.txt
```

Once you've fixed all merge conflicts, do:

```
git commit
```

Note: Advanced Git users may want to rebase instead of merge, but we squash and merge PRs either way.

5.3 Guidelines

- All code should have tests.
- All code should be documented, to the same [standard](#) as NumPy and SciPy.
- All changes are reviewed. Ask on the [mailing list](#) if you get no response to your pull request.

5.4 Testing

To run all tests:

```
$ PYTHONPATH=. pytest pygraphviz
```

Or tests from a specific file:

```
$ PYTHONPATH=. pytest pygraphviz/tests/test_readwrite.py
```

Use `--doctest-modules` to run doctests. For example, run all tests and all doctests using:

```
$ PYTHONPATH=. pytest --doctest-modules pygraphviz
```

Tests for a module should ideally cover all code in that module, i.e., statement coverage should be at 100%.

To measure the test coverage, run:

```
$ PYTHONPATH=. pytest --cov=pygraphviz pygraphviz
```

This will print a report with one line for each file in *pygraphviz*, detailing the test coverage:

Name	Stmts	Miss	Cover

pygraphviz/__init__.py	12	4	67%
pygraphviz/agraph.py	1022	196	81%
pygraphviz/graphviz.py	179	42	77%
pygraphviz/scrapper.py	26	18	31%
pygraphviz/testing.py	16	0	100%

TOTAL	1255	260	79%

5.5 Adding tests

If you're **new to testing**, see existing test files for examples of things to do. **Don't let the tests keep you from submitting your contribution!** If you're not sure how to do this or are having trouble, submit your pull request anyway. We will help you create the tests and sort out any kind of problem during code review.

5.6 Adding examples

The gallery examples are managed by [sphinx-gallery](#). The source files for the example gallery are `.py` scripts in `examples/` that generate one or more figures. They are executed automatically by `sphinx-gallery` when the documentation is built. The output is gathered and assembled into the gallery.

You can **add a new** plot by placing a new `.py` file in one of the directories inside the `examples` directory of the repository. See the other examples to get an idea for the format.

Note: Gallery examples should start with `plot_`, e.g. `plot_new_example.py`

General guidelines for making a good gallery plot:

- Examples should highlight a single feature/command.
- Try to make the example as simple as possible.
- Data needed by examples should be included in the same directory and the example script.
- Add comments to explain things are aren't obvious from reading the code.
- Describe the feature that you're showcasing and link to other relevant parts of the documentation.

5.7 Bugs

Please [report bugs on GitHub](#).

A

acyclic() (*AGraph method*), 22
 add_cycle() (*AGraph method*), 22
 add_edge() (*AGraph method*), 22
 add_edges_from() (*AGraph method*), 23
 add_node() (*AGraph method*), 23
 add_nodes_from() (*AGraph method*), 24
 add_path() (*AGraph method*), 24
 add_subgraph() (*AGraph method*), 24
 AGraph (*class in pygraphviz*), 19

C

clear() (*AGraph method*), 24
 close() (*AGraph method*), 24
 copy() (*AGraph method*), 24

D

degree() (*AGraph method*), 24
 degree_iter() (*AGraph method*), 24
 delete_edge() (*AGraph method*), 24
 delete_edges_from() (*AGraph method*), 25
 delete_node() (*AGraph method*), 25
 delete_nodes_from() (*AGraph method*), 25
 delete_subgraph() (*AGraph method*), 25
 directed (*AGraph property*), 25
 draw() (*AGraph method*), 25

E

edges() (*AGraph method*), 26
 edges_iter() (*AGraph method*), 26

F

from_string() (*AGraph method*), 26

G

get_edge() (*AGraph method*), 26
 get_name() (*AGraph method*), 27
 get_node() (*AGraph method*), 27
 get_subgraph() (*AGraph method*), 27

H

has_edge() (*AGraph method*), 27

has_neighbor() (*AGraph method*), 27
 has_node() (*AGraph method*), 27

I

in_degree() (*AGraph method*), 28
 in_degree_iter() (*AGraph method*), 28
 in_edges() (*AGraph method*), 28
 in_edges_iter() (*AGraph method*), 28
 in_neighbors() (*AGraph method*), 28
 is_directed() (*AGraph method*), 28
 is_strict() (*AGraph method*), 28
 is_undirected() (*AGraph method*), 28
 iterdegree() (*AGraph method*), 28
 iteredges() (*AGraph method*), 28
 iterindegree() (*AGraph method*), 28
 iterinedges() (*AGraph method*), 28
 iterneighbors() (*AGraph method*), 29
 iternodes() (*AGraph method*), 29
 iteroutdegree() (*AGraph method*), 29
 iteroutedges() (*AGraph method*), 29
 iterpred() (*AGraph method*), 29
 itersucc() (*AGraph method*), 29

L

layout() (*AGraph method*), 29

N

name (*AGraph property*), 30
 neighbors() (*AGraph method*), 30
 neighbors_iter() (*AGraph method*), 30
 nodes() (*AGraph method*), 30
 nodes_iter() (*AGraph method*), 30
 number_of_edges() (*AGraph method*), 30
 number_of_nodes() (*AGraph method*), 30

O

order() (*AGraph method*), 30
 out_degree() (*AGraph method*), 30
 out_degree_iter() (*AGraph method*), 30
 out_edges() (*AGraph method*), 30
 out_edges_iter() (*AGraph method*), 30

`out_neighbors()` (*AGraph method*), 30

P

`predecessors()` (*AGraph method*), 30

`predecessors_iter()` (*AGraph method*), 30

R

`read()` (*AGraph method*), 31

`remove_edge()` (*AGraph method*), 31

`remove_edges_from()` (*AGraph method*), 31

`remove_node()` (*AGraph method*), 31

`remove_nodes_from()` (*AGraph method*), 31

`remove_subgraph()` (*AGraph method*), 31

`reverse()` (*AGraph method*), 31

S

`strict` (*AGraph property*), 31

`string()` (*AGraph method*), 31

`string_nop()` (*AGraph method*), 31

`subgraph()` (*AGraph method*), 32

`subgraph_parent()` (*AGraph method*), 32

`subgraph_root()` (*AGraph method*), 32

`subgraphs()` (*AGraph method*), 32

`subgraphs_iter()` (*AGraph method*), 32

`successors()` (*AGraph method*), 32

`successors_iter()` (*AGraph method*), 32

T

`to_directed()` (*AGraph method*), 32

`to_string()` (*AGraph method*), 32

`to_undirected()` (*AGraph method*), 32

`tred()` (*AGraph method*), 32

U

`unflatten()` (*AGraph method*), 32

W

`write()` (*AGraph method*), 33