PyGraphviz Documentation

Release 1.1

Aric Hagberg, Dan Schult

May 14, 2011

CONTENTS

CHAPTER

ONE

INSTALLING

1.1 Quick Install

Get PyGraphviz from the Python Package Index at http://pypi.python.org/pypi/pygraphviz

or install it with:

```
easy_install pygraphviz
```

and an attempt will be made to find and install an appropriate version that matches your operating system and Python version.

More download options are at http://networkx.lanl.gov/download.html

1.2 Installing from Source

You can install from source by downloading a source archive file (tar.gz or zip) or by checking out the source files from the Subversion repository.

1.2.1 Source Archive File

1. Download the source (tar.gz or zip file).
2. Unpack and change directory to pygraphviz-"version"
3. Run "python setup.py install" to build and install
4. (optional) Run "python setup_egg.py nosetests" to execute the tests

1.2.2 SVN Repository

1. Check out the PyGraphviz trunk:

   ```
   svn co https://networkx.lanl.gov/svn/pygraphviz/trunk pygraphviz
   ```

2. Change directory to "pygraphviz"
3. Run "python setup.py install" to build and install
4. (optional) Run "python setup_egg.py nosetests" to execute the tests

If you don't have permission to install software on your system, you can install into another directory using the –prefix or –home flags to setup.py.

For example

```
python setup.py install --prefix=/home/username/python
or
python setup.py install --home=~
```

If you didn't install in the standard Python site-packages directory you will need to set your PYTHONPATH variable to the alternate location. See http://docs.python.org/inst/search-path.html for further details.

1.3 Requirements

1.3.1 GraphViz

To use PyGraphviz you need GraphViz version 2.16 or later. Some versions have known bugs that have been fixed; get the latest release available for best results.

- Official site: http://www.graphviz.org

1.3.2 Python

To use PyGraphviz you need Python version 2.4 or later http://www.python.org/

The easiest way to get Python and most optional packages is to install the Enthought Python distribution http://www.enthought.com/products/epd.php

Other options are

Windows

- Official Python site version: http://www.python.org/download/
- ActiveState version: http://activestate.com/Products/ActivePython/

OSX

OSX 10.5 ships with Python version 2.5. If you have an older version we encourage you to download a newer release. Pre-built Python packages are available from

- Official Python site version http://www.python.org/download/
- Pythonmac http://www.pythonmac.org/packages/
- ActiveState http://activestate.com/Products/ActivePython/

If you are using Fink or MacPorts, Python is available through both of those package systems.

Linux

Python is included in all major Linux distributions

CHAPTER

TWO

TUTORIAL

The API is very similar to that of NetworkX. Much of the NetworkX tutorial at http://networkx.lanl.gov/tutorial/ is applicable to PyGraphviz. See https://networkx.lanl.gov/pygraphviz/reference/api_notes.html for major differences.

2.1 Start-up

Import PyGraphviz with

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

or to bring into the current namespace without the "pgv" prefix

```
>>> from pygraphviz import *
```

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 intializing 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
```

CHAPTER

THREE

REFERENCE

3.1 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

```
>>> from pygraphviz import *
>>> G=AGraph()
>>> G=AGraph(directed=True)
>>> G=AGraph("file.dot")
```

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

```
>>> G=AGraph(landscape='true',ranksep='0.1')
```

or alternatively

```
>>> G=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 automaticaly detect the type so you may write for example:

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

string:  String containing a dot format graph

handle:  Swig pointer to an agraph_t data structure
```

acyclic(*args=''*, *copy=False*)
Reverse sufficient edges in digraph to make graph acyclic. Modifies existing graph.

To create a new graph use

```
>>> A=AGraph()
>>> 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 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).

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

Attributes can be added when edges are created

```
>>> G.add_edge(u'a',u'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.

```
>>> G=AGraph()
>>> elist=[('a','b'),('b','c')]
>>> G.add_edges_from(elist)
```

Attributes can be added when edges are created

```
>>> 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).

```
>>> G=AGraph()
>>> G.add_node('a')
>>> G.nodes()
[u'a']
>>> G.add_node(1) # will be converted to a string
>>> G.nodes()
[u'a', u'1']
```

Attributes can be added to nodes on 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

```
>>> G=AGraph()
>>> nlist=['a','b',1,'spam']
>>> G.add_nodes_from(nlist)
>>> sorted(G.nodes())
[u'1', u'a', u'b', u'spam']
```

Attributes can be added to nodes on 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.

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 paris 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.

```
>>> G=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

```
>>> G=AGraph()
>>> nlist=['a','b',1,'spam']
>>> G.add_nodes_from(nlist)
>>> G.remove_nodes_from(nlist)
```

delete_subgraph(*name*)
Remove subgraph with given name.

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'.

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.

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

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

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.ps',prog='twopi',args='-Gepsilon=1')

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.

```
>>> G=AGraph()
>>> G.add_edge('a','b')
>>> G.add_edge('c','d')
>>> print sorted(G.edges())
[(u'a', u'b'), (u'c', u'd')]
>>> print G.edges('a')
[(u'a', u'b')]
```

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.

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

```
>>> s='digraph {1 -> 2}'
>>> A=AGraph()
>>> t=A.from_string(s)
>>> A=AGraph(string=s) # specify s is a string
>>> A=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).

```
>>> G=AGraph()
>>> G.add_edge('a','b')
>>> edge=G.get_edge('a','b')
>>> print edge
(u'a', u'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.

```
>>> G=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.

```
>>> G=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.

```
>>> G=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.

```
>>> G=AGraph()
>>> G.add_node('a')
>>> G.has_node('a')
True
>>> '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 paris 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 paris of (node,degree).

iteredges(*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.

iterindegree(*nbunch=None*)
Return an iterator over the in-degree of the nodes given in nbunch container.

Returns paris 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 paris 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.

```
>>> A=AGraph()
>>> A.layout() # uses neato
>>> A.layout(prog='dot')
```

Use keyword args to add additional arguments to graphviz programs.

The layout might take a long time on large graphs.

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 paris 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.

prepare_nbunch(*nbunch=None*)

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.

```
>>> G=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

```
>>> G=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.

size()
Return the number of edges in the graph.

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) represetnation 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(***kwds*)
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 (unicode) representation of graph in dot format.

to_undirected()
Return undirected copy of graph.

tred(*args=''*, *copy=False*)
Transitive reduction of graph. Modifies existing graph.

To create a new graph use

```
>>> A=AGraph()
>>> B=A.tred(copy=True)
```

See the graphviz "tred" program for details of the algorithm.

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')
```

3.2 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. You can

1. set the LD_LIBRARY_PATH environment variable e.g. export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/usr/lib/graphviz
2. configure your system with the additional path. e.g. for Linux add a line to /etc/ld.so.conf and run ldconfig

Q How do I compile pygraphviz under Windows? And why don't you distribute a pygraphviz Windows installer?

A We don't have Windows development machines but would like to have pygraphviz work on all platforms. If you have success with Windows or would be willing to help test and distribute a Windows installer please drop us a note.

See also tickets: **https://networkx.lanl.gov/trac/ticket/117**
https://networkx.lanl.gov/trac/ticket/491

3.3 API Notes

3.3.1 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

3.3.2 pygraphivz-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 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. The are primarily used for clustering of nodes when drawing with dot.

Pygraphviz supports most of the Graphviz API.

3.4 News

3.4.1 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

3.4.2 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

3.4.3 pygraphviz-0.99.1

Release date: 7 December 2008

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

- Use Graphviz libcgraph 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

3.4.4 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

3.4.5 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/do
  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 stings 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

3.4.6 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')

3.4.7 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)

3.4.8 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

3.4.9 pygraphviz-0.33

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

3.4.10 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

3.5 Related Pacakges

- 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/

3.6 History

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

3.7 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.

3.8 Legal

3.8.1 PyGraphviz License

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.

3.8.2 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.

CHAPTER

FOUR

EXAMPLES

See the examples for sample usage and ideas https://networkx.lanl.gov/trac/browser/pygraphviz/trunk/examples/
There is a complete reference guide at https://networkx.lanl.gov/pygraphviz/reference

4.1 Simple

A basic example showing how to read and write dot files and draw graphs.
https://networkx.lanl.gov/trac/browser/pygraphviz/trunk//examples/simple.py

4.2 Star

An example showing how to set attributes.
https://networkx.lanl.gov/trac/browser/pygraphviz/trunk/examples/star.py

4.3 Miles

An example showing how to use Graphviz to draw a graph with given positions.
https://networkx.lanl.gov/trac/browser/pygraphviz/trunk/examples/miles.py

INDEX

P

R

S

T

W