NetworkX Tutorial

last updated: 11 April 2005

Home Downloads News Tutorial Reference QuickRef Examples Drawing Screenshots MailingList Developers Credits Legal

Introduction

NetworkX = Network "X" = NX (for short)

Original Creators:


```
Aric Hagberg, hagberg@lanl.gov
Pieter Swart, swart@lanl.gov
```


NetworkX is a Python-based package for the creation, manipulation, and study of the structure, dynamics, and function of complex networks. The name means **Network "X"** and we pronounce it **NX**. We will refer to (and import) NetworkX as NX for the sake of brevity.

The structure of a graph or network is encoded in the **edges** (connections, links, ties, arcs, bonds) between **nodes** (vertices, sites, actors). If unqualified, by graph we mean a simple undirected graph, i.e. no self-loops and no multiple edges are allowed. By a network we usually mean a graph with weights (fields, properties) on nodes and/or edges.

The potential audience for NetworkX include: mathematicians, physicists, biologists, computer scientists, social scientists. The current state of the art of the (young and rapidly growing) science of complex networks is presented in Albert and Barabasi [BA02], Newman [Newman03], and Dorogovtsev and Mendes [DM03]. See also the classic texts [Bollobas01], [Diestel97] and [West01] for graph theoretic results and terminology. For basic graph algorithms, we recommend the texts of Sedgewick, e.g. [Sedgewick01] and [Sedgewick02].

Why Python? Past experience showed this approach to maximize productivity, power, multidisciplinary scope (our application test beds included large communication, social, data and biological networks), and platform independence (although we have only extensively tested it under Linux). This philosophy does not exclude using whatever other language is appropriate for a specific subtask, since Python is also an excellent "glue" language [Langtangen04]. Equally important, Python is free, well-supported and a joy to use. Among the many guides to python, we recommend the documentation at www.python.org and the text by Alex Martelli [Martelli03].

Obtaining and Installing NetworkX

You need Python. We recommend the latest stable release available from http://www.python.org/. The latest version can be found at http://networkx.lanl.net/. NX will work on multiple platforms.

On Linux platforms, download the current tarball numbered, say, networkx-x.y.z.tar.gz, to an appropriate directory,say /home/username/networks

```
gzip -d -c networkx-x.y.z.tar.gz|tar xvf-
cd networkxx-x.y.z
# do the following with your preferred python version
# if you are using cvs, remove your build directory first
python setup.py build
# change to an id, that is allowed to do installation
python setup.py install
```

This will install NetworkX in your python site-packages directory.

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.

A Quick Tour

Building and drawing a small graph

We assume you can start an interactive python session. At the time of writing we require Python Release 2.3.4 or later. (Although not required, if you want to run the unit tests you will need Release 2.4 or later.) We will assume that you are familiar with Python terminology (see the official python website http://www.python.org for more information).

```
%python
Python 2.3.4 (#1, Jun  3 2004, 14:57\:21)
[GCC 3.2.2 20030222 (Red Hat Linux 3.2.2-5)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

(It might be useful to add the home directory where NX is installed to your PYTHONPATH.)

After starting python, import the networkx module with (the recommended way)

```
>>> import networkx as NX
```

or (the usual mode for interactive experimentation that could clobber some names already in your namespace)

```
>>> from networkx import *
```

(If this import fails, it means that python cannot find the installed module. Check your installation and your PYTHONPATH.)

To save much repetition, in all the examples below we will assume that NX has been imported with

```
>>> from networkx import *
```

To create a new (simple) graph, call Graph() with zero or more arguments.

```
>>> G=Graph()
```

When called with zero arguments, one obtains the empty graph without any nodes or edges. In NX every graph or network is a python "object", and in python the functions associated with an "object" are known as methods.

The following classes are provided:

Graph The basic operations common to graph-like classes. This class implements a "simple graph"; it ignores multiple edges between two nodes and does not allow edges from a node to itself (self-loops).

DiGraph Operations common to digraphs, simple graphs with directed edges. (A subclass of Graph.)

XGraph A flexible (and experimental) graph class that allows data/weights/labels/objects to be associated with each edge. While a simple graph by default, this class can also allow multiple edges and self loops. Thus, it can be used to represent a weighted graph, pseudograph, or network. This additional flexibility leads to some degradation in performance, though usually not significant. (A subclass of Graph.)

XDiGraph A directed version of an XGraph. (A subclass of DiGraph.)

Empty graph-like objects are created with

- G=Graph()
- G=DiGraph()
- G=XGraph()
- G=XGraph(selfloops=True, multiedges=True)
- G=XDiGraph(selfloops=True, multiedges=True)

This package implements graphs using data structures based on an adjacency list implemented as a node_centric dictionary of dictionaries. The dictionary contains keys corresponding to the nodes and the values are dictionaries of neighboring node keys with the value 1 (or edge data for XGraph(), or a list of edge data for XGraph(multiedges=True)). This allows fast addition, deletion and lookup of nodes and neighbors in large graphs. The underlying datastructure should only be visible in the modules base.py and xbase.py. In all other modules, graph-like objects are manipulated solely via the methods defined in base.py and xbase.py, and not by acting directly on the datastructure.

The following shorthand is used throughout NetworkX documentation and code: (we use mathematical notation n,v,w,... to indicate a node, v=vertex=node).

G,G1,G2,H,etc: Graphs

n,n1,n2,u,v,v1,v2: nodes (vertices)

nlist,vlist: a list of nodes (vertices)

nbunch, vbunch: a "bunch" of nodes (vertices). an nbunch is any iterable container of nodes that is not itself a node in the graph. (It can be an iterable or an iterator, e.g. a list, set, graph, file, etc..)

e=(n1,n2): an edge (a python 2-tuple) in Graph and DiGraph, also written n1-n2 (if undirected) and n1->n2 (if directed).

e=(n1,n2,x): an edge (a python 3-tuple) in XGraph and XDiGraph, containing the two nodes connected and the edge data/label/object stored associated with the edge. The object x, or a list of objects (if multiedges=True), can be obtained using G.get_edge(n1,n2). In XGraph G.add_edge(n1,n2) is equivalent to G.add_edge(n1,n2,1). However, G.delete_edge(n1,n2) will delete all edges between n1 and n2.

elist: a list of edges (as 2- or 3-tuples)

ebunch: a bunch of edges (as tuples) an ebunch is any iterable (non-string) container of edge-tuples. (Similar to nbunch, also see add_edge).

Warning:
- The ordering of objects within an arbitrary nbunch/ebunch can be machine- or implementation-dependent.
- Algorithms applicable to arbitrary nbunch/ebunch should treat them as once-through-and-exhausted iterable containers.
- len(nbunch) and len(ebunch) need not be defined.

Graph methods

A Graph object G has the following primitive methods associated with it:
(You can use dir(G) to inspect the methods associated with object G.)

1. Non-mutating Graph methods:

   - len(G) number of nodes in G
   - G.has_node(n)
   - n in G (equivalent to G.has_node(n))
   - G.nodes()
   - G.nodes_iter()
   - G.has_edge(n1,n2)
   - G.edges(), G.edges(n), G.edges(nbunch)
   - G.edges_iter(), G.edges_iter(n), G.edges_iter(nbunch)
   - G.neighbors(n)
   - G[n] (equivalent to G.neighbors(n))
   - G.neighbors_iter(n) # iterator over neighbors
   - G.number_of_nodes()
   - G.number_of_edges()
   - G.node_boundary(nbunch)
   - G.node_boundary(nbunch1,nbunch2)
   - G.edge_boundary(nbunch)
   - G.edge_boundary(nbunch1,nbunch2)
   - G.degree(n), G.degree(nbunch)
   - G.degree_iter(n), G.degree_iter(nbunch)
   - G.is_directed()

   The following return a new graph:

   - G.subgraph(nbunch)
   - G.subgraph(nbunch, create_using=H)
   - G.copy()
   - G.to_directed()
   - G.to_undirected()

2. Mutating Graph methods:

   - G.add_node(n), G.add_nodes_from(nbunch)
   - G.delete_node(n), G.delete_nodes_from(nbunch)
   - G.add_edge(n1,n2), G.add_edge(e)
   - G.add_edges_from(ebunch)

- G.delete_edge(n1,n2), G.delete_edge(e),
- G.delete_edges_from(ebunch)
- G.add_path(nlist)
- G.add_cycle(nlist)
- G.clear()
- G.subgraph(nbunch,inplace=True)

Names of classes/objects use the CapWords convention, e.g. Graph, XDiGraph. Names of functions and methods use the lowercase_words_separated_by_underscores convention, e.g. petersen_graph(), G.add_node(10).

G can be inspected interactively by typing "G" (without the quotes). This will reply something like <networkx.base.Graph object at 0x40179a0c>. (On linux machines with CPython the hexadecimal address is the memory location of the object.)

Examples

Create an empty graph with zero nodes and zero edges.

```
>>> from networkx import *
>>> G=Graph()
```

G can be grown in several ways. By adding one node at a time:

```
>>> G.add_node(1)
```

by adding a list of nodes:

```
>>> G.add_nodes_from([2,3])
```

or by adding any nbunch of nodes (see above definition of an nbunch):

```
>>> H=path_graph(10)
>>> G.add_nodes_from( H )
```

(H can be a graph, iterator, string, set, or even a file.)

Any hashable object can represent a node, e.g. a Graph, a customized node object, etc.

```
>>> G.add_node(H)
```

(You should not change the object if the hash depends on its contents.)

G can also be grown by adding one edge at a time:

```
>>> G.add_edge( (1,2) )
```

by adding a list of edges:

```
>>> G.add_edges_from([(1,2),(1,3)])
```

or by adding any ebunch of edges (see above definition of an ebunch):

```
>>> G.add_edges_from(H.edges())
```

One can demolish the graph in a similar fashion; using delete_node, delete_nodes_from, delete_edge and delete_edges_from, e.g.

```
>>> G.delete_node(H)
```

There are no complaints when adding existing nodes or edges. For example: after removing all nodes and edges,

```
>>> G.clear()
>>> G.add_edges_from([(1,2),(1,3)])
```

will add new nodes as required.

```
>>> G.add_node("spam")
```

At this stage the graph G consists of 4 nodes and 2 edges, as can be seen by:

```
>>> number_of_nodes(G)
4
>>> number_of_edges(G)
2
```

we can examine them with:

```
>>> G.nodes()
[1, 2, 3, 'spam']
>>> G.edges()
[(1, 2), (1, 3)]
```

Drawing a small graph

NetworkX does not provide sophisticated graph drawing tools. We do provide elementary drawing tools as well as an interface to use the open source Graphviz software package. These reside in networkx.drawing, and will be imported if possible. See the Drawing section for details.

```
>>> from networkx.drawing import *
```

To test if this import was successful draw G using one of:

```
>>> draw(G)
>>> draw_random(G)
>>> draw_circular(G)
>>> draw_spectral(G)
```

when drawing to an interactive display. Note that you may need to issue a

```
>>> show()
```

if you are not using matplotlib in interactive mode (http://matplotlib.sourceforge.net/faq.html#SHOW). Or use

```
>>> draw(G)
>>> savefig("path.ps")
```

to write to the file "path.ps" in the local directory. If graphviz and pydot are available on your system, you can also use:

```
>>> draw_nxpydot(G)
>>> write_dot(G)
graph G {
"1";
"2";
"3";
"spam";
"1" -- "2";
"1" -- "3";
}
<BLANKLINE>
```

You may find it useful to interactively test code using "ipython -pylab", thereby combining the power of ipython and matplotlib.

Functions for analyzing graph properties

The structure of G can be analyzed using various graph theoretic functions such as:

```
>>> connected_components(G)
[[1, 2, 3], ['spam']]

>>> diameter(G)
2

>>> sorted(degree(G))
[0, 1, 1, 2]

>>> clustering(G)
[0.0, 0.0, 0.0, 0.0]

>>> sorted(eccentricity(G))
[0, 1, 2, 2]
```

Some functions defined on the nodes, e.g. degree() and clustering(), can be given a single node or an nbunch of nodes as argument. If a single node is specified, then a single value is returned. If an iterable nbunch is specified, then the function will return a list of values. With no argument, the function will return a list of values at all nodes of the graph.

```
>>> degree(G,1)
2
>>> G.degree(1)
2

>>> sorted(degree(G,[1,2]))
[1, 2]

>>> sorted(degree(G))
[0, 1, 1, 2]
```

When called with the "with_labels"=True option a dict with nodes as keys and function values as arguments is returned.

```
>>> degree(G,[1,2],with_labels=True)
{1: 2, 2: 1}
>>> degree(G,with_labels=True)
{1: 2, 2: 1, 3: 1, 'spam': 0}
```

Graph generators and graph operations

In addition to constructing graphs node-by-node or edge-by-edge, they can also be generated by:

1. Applying classic graph operations, such as:

```
subgraph(G, nbunch)      - induce subgraph of G on nodes in nbunch
union(G1,G2)             - graph union
disjoint_union(G1,G2)    - graph union assuming all nodes are different
cartesian_product(G1,G2) - return Cartesian product graph
compose(G1,G2)           - combine graphs identifying nodes common
                           to both
complement(G)            - graph complement
create_empty_copy(G)     - return an empty copy of the same graph class
convert_to_undirected(G) - return an undirected representation of G
convert_to_directed(G)   - return a directed representation of G
```

2. Using a call to one of the classic small graphs, e.g.

```
>>> petersen=petersen_graph()
>>> tutte=tutte_graph()
>>> maze=sedgewick_maze_graph()
>>> tet=tetrahedral_graph()
```

3. Using a (constructive) generator for a classic graph, e.g.

```
>>> K_5=complete_graph(5)
>>> K_3_5=complete_bipartite_graph(3,5)
>>> barbell=barbell_graph(10,10)
>>> lollipop=lollipop_graph(10,20)
```

4. Using a stochastic graph generator, e.g.

```
>>> er=erdos_renyi_graph(100,10)
>>> ws=watts_strogatz_graph(30,3,0.1)
>>> ba=barabasi_albert_graph(100,5)
>>> red=random_lobster(100,0.9,0.9)
```

Graph IO

Reading a graph from a file

```
>>> G=tetrahedral_graph()
```

Write to adjacency list format

```
>>> write_adjlist(G, "tetrahedral.adjlist")
```

Read from adjacency list format

```
>>> H=read_adjlist("tetrahedral.adjlist")
```

Write to edge list format

```
>>> write_edgelist(G, "tetrahedral.edgelist")
```

Read from edge list format

```
>>> H=read_edgelist("tetrahedral.edgelist")
```

See also Interfacing with other tools below for how to draw graphs with matplotlib or graphviz.

Graphs with multiple edges and self-loops

See the XGraph and XDiGraph classes. For example, to build Euler's famous graph of the bridges of Konigsberg over the Pregel river, one can use:

```
>>> K=XGraph(name="Konigsberg", multiedges=True, selfloops=False)
>>> K.add_edges_from([("A","B","Honey Bridge"),
...                   ("A","B","Blacksmith's Bridge"),
...                   ("A","C","Green Bridge"),
...                   ("A","C","Connecting Bridge"),
...                   ("A","D","Merchant's Bridge"),
...                   ("C","D","High Bridge"),
...                   ("B","D","Wooden Bridge")])
>>> K.degree("A")
5
```

Directed Graphs

The DiGraph class provides operations common to digraphs (graphs with directed edges). A subclass of Graph, Digraph adds the following methods to those of Graph:

- successors
- successors_iter
- predecessors
- predecessors_iter
- out_degree
- out_degree_iter
- in_degree
- in_degree_iter

See networkx.DiGraph for more documentation.

Interfacing with other tools

NetworkX provides interfaces to matplotlib and graphviz for graph layout (node and edge positioning) and drawing. We also use matplotlib for graph spectra and in some drawing operations. Without either, one can still use the basic graph-related functions.

See the graph Drawing section for details on how to install and use these tools.

Matplotlib

```
>>> G=tetrahedral_graph()
>>> draw(G)
```

Graphviz

```
>>> G=tetrahedral_graph()
>>> write_dot(G,"tetrahedral.dot")
```

Specialized Topics

Graphs composed of general objects

For most applications, nodes will have string or integer labels (as in the example above). The power of Python ("everything is an object") allows us to construct graphs with ANY hashable object as a node. (Note though that this will not work with non-python datastructures, e.g. building a graph on a wrapped Python version of graphviz).

For example, one can construct a graph with Python mathematical functions as nodes, and where two mathematical functions are connected if they are in the same chapter in some Handbook of Mathematical Functions. E.g.

```
>>> from math import *
>>> G=Graph()
>>> G.add_node(acos)
>>> G.add_node(sinh)
>>> G.add_node(cos)
```

```
>>> G.add_node(tanh)
>>> G.add_edge(acos,cos)
>>> G.add_edge(sinh,tanh)
>>> sorted(G.nodes())
[<built-in function acos>, <built-in function cos>, <built-in function sinh>, <built-in function tan
```

As another example, one can build (meta) graphs using other graphs as the nodes.

We have found this power quite useful, but hasten to add that its abuse can lead to unexpected surprises unless one is familiar with Python. If in doubt, the user should use convert_node_labels_to_integers to obtain a more traditional graph.

Imbedding general objects onto edges

An XGraph and XDiGraph object allows associating arbitrary objects with an edge. In these classes edges are 3-tuples (n1,n2,x), representing an edge between nodes n1 and n2 that is decorated with the object x (not necessarily hashable). For example, n1 and n2 can be protein objects from the RCSB Protein Data Bank, and x can refer to an XML record of a publication detailing experimental observations of their interaction. These classes are still in the experimental stage, with not all the graph-related functions and operations tested on them. Use with caution and tell us if you find them useful.

Unit tests

For most modules, say base.py, the command "python base.py" will run several unit tests in the networkx/tests subdirectory. This requires the use of Python 2.4 or later. To run all the unit tests, run "python test.py" in the networkx/tests subdirectory.

Not everything is an object

NX developed from the need to analyze dynamics on a diverse collection of large networks and we have stubbornly refused to objectify all the mathematical structures of graph theory down to the atomic level. Neither nodes nor edges are objects. A node can be any hashable object, and an edge is a 2-tuple (n1,n2) of nodes (in the case of Graph and DiGraph) or a triple (n1,n2,x) (in the case of XGraph and XDiGraph) consisting of two nodes and an object x decorating that edge.

Graph dna

Some basic properties of a graph are stored in a dictionary called the graph dna (graph dna = graph properties). These include the name, the datastructure, and possibly some other properties. The graph dna is provided as a user-defined variable and should not be relied on.

Use print_dna() to inspect the current dna of graph G, e.g.

```
>>> G=circular_ladder_graph(20)
>>> G.print_dna()
datastructure  : vdict_of_dicts
```

References

[BA02] R. Albert and A.-L. Barabasi, "Statistical mechanics of complex networks", Reviews of Modern Physics, 74, pp. 47-97, 2002. (Preprint available online at http://citeseer.ist.psu.edu/442178.html or http://arxiv.org/abs/cond-mat/0106096)

[Bollobas01] B. Bollobas, "Random Graphs", Second Edition, Cambridge University Press, 2001.

[Diestel97] R. Diestel, "Graph Theory", Springer-Verlag, 1997. (A free electronic version is available at http://www.math.uni-hamburg.de/home/diestel/books/graph.theory/download.html)

[DM03] S.N. Dorogovtsev and J.F.F. Mendes, "Evolution of Networks", Oxford University Press, 2003.

[Langtangen04] H.P. Langtangen, "Python Scripting for Computational Science.", Springer Verlag Series in Computational Science and Engineering, 2004.

[Martelli03] A. Martelli, "Python in a Nutshell", O'Reilly Media Inc, 2003. (A useful guide to the language is available at http://www.oreilly.com/catalog/pythonian/chapter/ch04.pdf)

[Newman03] M.E.J. Newman, "The Structure and Function of Complex Networks", SIAM Review, 45, pp. 167-256, 2003. (Available online at http://epubs.siam.org/sam-bin/dbq/article/42480)

[Sedgewick02] R. Sedgewick, "Algorithms in C: Parts 1-4: Fundamentals, Data Structure, Sorting, Searching", Addison Wesley Professional, 3rd ed., 2002.

[Sedgewick01] R. Sedgewick, "Algorithms in C, Part 5: Graph Algorithms", Addison Wesley Professional, 3rd ed., 2001.

[West01] D. B. West, "Introduction to Graph Theory", Prentice Hall, 2nd ed., 2001.