NetworkX Tutorial

Release 1.0.1

Aric Hagberg, Dan Schult, Pieter Swart

January 11, 2010

Contents

1 Creating a graph

Create an empty graph with no nodes and no edges.

```
>>> import networkx as nx
>>> G=nx.Graph()
```

By definition, a `Graph` is a collection of nodes (vertices) along with identified pairs of nodes (called edges, links, etc). In NetworkX, nodes can be any hashable object e.g. a text string, an image, an XML object, another Graph, a customized node object, etc. (Note: Python's None object should not be used as a node as it determines whether optional function arguments have been assigned in many functions.)

2 Nodes

The graph G can be grown in several ways. NetworkX includes many graph generator functions and facilities to read and write graphs in many formats. To get started though we'll look at simple manipulations. You can add one node at a time,

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

add a list of nodes,

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

or add any *nbunch* of nodes. An *nbunch* is any iterable container of nodes that is not itself a node in the graph. (e.g. a list, set, graph, file, etc..)

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

Note that G now contains the nodes of H as nodes of G. In contrast, you could use the graph H as a node in G.

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

The graph G now contains H as a node. This flexibility is very powerful as it allows graphs of graphs, graphs of files, graphs of functions and much more. It is worth thinking about how to structure your application so that the nodes are useful entities. Of course you can always use a unique identifier in G and have a separate dictionary keyed by identifier to the node information if you prefer. (Note: You should not change the node object if the hash depends on its contents.)

3 Edges

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

```
>>> G.add_edge(1,2)
>>> e=(2,3)
>>> G.add_edge(*e) # unpack edge tuple*
```

by adding a list of edges,

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

or by adding any *ebunch* of edges, An *ebunch* is any iterable container of edge-tuples. An edge-tuple can be a 2-tuple of nodes or a 3-tuple with 2 nodes followed by an edge attribute dictionary, e.g. (2,3,{'weight':3.1415}). Edge attributes are discussed further below

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

One can demolish the graph in a similar fashion; using `Graph.remove_node()`, `Graph.remove_nodes_from()`, `Graph.remove_edge()` and `Graph.remove_edges_from()`, e.g.

```
>>> G.remove_node(H)
```

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

```
>>> G.clear()
```

we add new nodes/edges and NetworkX quietly ignores any that are already present.

```
>>> G.add_edges_from([(1,2),(1,3)])
>>> G.add_node(1)
>>> G.add_edge(1,2)
>>> G.add_node("spam")        # adds node "spam"
>>> G.add_nodes_from("spam") # adds 4 nodes: 's', 'p', 'a', 'm'
```

At this stage the graph G consists of 8 nodes and 2 edges, as can be seen by:

```
>>> G.number_of_nodes()
8
>>> G.number_of_edges()
2
```

We can examine them with

```
>>> G.nodes()
['a', 1, 2, 3, 'spam', 'm', 'p', 's']
>>> G.edges()
[(1, 2), (1, 3)]
>>> G.neighbors(1)
[2, 3]
```

Removing nodes or edges has similar syntax to adding:

```
>>> G.remove_nodes_from("spam")
>>> G.nodes()
[1, 2, 3, 'spam']
>>> G.remove_edge(1,3)
```

You can specify graph data upon instantiation if an appropriate structure exists.

```
>>> H=nx.DiGraph(G)    # create a DiGraph using the connections from G
>>> H.edges()
[(1, 2), (2, 1)]
>>> H=nx.Graph({0:[1,2,3], 1:[0,3], 2:[0], 3:[0]})  # dict-of-lists adjacency
```

4 Edge Objects

Edge data/weights/labels/objects can also be associated with an edge. Each edge has an attribute dictionary associated with it. Arbitrary key=value attributes can be assigned. The special attribute 'weight' should be numeric and holds values used by algorithms requiring weighted edges.

```
>>> H=nx.Graph()
>>> H.add_edge(1,2,color='red')
>>> H.add_edges_from([(1,3,{'color':'blue'}), (2,0,{'color':'red'}), (0,3)])
>>> H.edges()
[(0, 2), (0, 3), (1, 2), (1, 3)]
>>> H.edges(data=True)
[(0, 2, {'color': 'red'}), (0, 3, {}), (1, 2, {'color': 'red'}), (1, 3, {'color': 'blue'})]
```

To update the edge attributes for an existing edge, add the edge again with the new value. (Note: with `MultiGraph` you need to keep track of the edge key for the edge you want to update.)

```
>>> H.add_edge(0,2,color='blue')
>>> H.edges(data=True)
[(0, 2, {'color': 'blue'}), (0, 3, {}), (1, 2, {'color': 'red'}), (1, 3, {'color': 'blue'})]
```

You might notice that nodes and edges are not NetworkX objects. This leaves you free to use your existing node and edge objects, or more typically, use numerical values or strings where appropriate. A node can be any hashable object (except None), and an edge can be associated with any object x using G.add_edge(n1,n2,object=x).

As an example, n1 and n2 could be protein objects from the RCSB Protein Data Bank, and x could refer to an XML record of publications detailing experimental observations of their interaction.

We have found this power quite useful, but its abuse can lead to unexpected surprises unless one is familiar with Python. If in doubt, consider using `convert_node_labels_to_integers()` to obtain a more traditional graph with integer labels.

5 Accessing edges

In addition to the methods `Graph.nodes()`, `Graph.edges()`, and `Graph.neighbors()`, iterator versions (e.g. `Graph.edges_iter()`) can save you from creating large lists when you are just going to iterate through them anyway.

Fast direct access to the graph data structure is also possible using subscript notation.

> **Warning:** Do not change the returned dict–it is part of the graph data structure and direct manipulation may leave the graph in an inconsistent state.

```
>>> G[1]  # Warning: do not change the resulting dict
{2: {}}
>>> G[1][2]
{}
```

You can safely set the attributes of an edge using subscript notation if the edge aleady exists.

```
>>> G.add_edge(1,3)
>>> G[1][3]['color']='blue'
```

Fast examination of all edges is achieved using adjacency iterators. Note that for undirected graphs this actually looks at each edge twice.

```
>>> FG=nx.Graph()
>>> FG.add_weighted_edges_from([(1,2,0.125),(1,3,0.75),(2,4,1.2),(3,4,0.375)])
>>> for n,nbrs in FG.adjacency_iter():
...    for nbr,eattr in nbrs.iteritems():
...        data=eattr['weight']
...        if data<0.5: print (n,nbr,data)
(1, 2, 0.125)
(2, 1, 0.125)
(3, 4, 0.375)
(4, 3, 0.375)
```

6 Directed graphs

The DiGraph class provides additional methods specific to directed edges, e.g. `DiGraph.out_edges()`, `DiGraph.in_degree()`, `DiGraph.predecessors()`, `DiGraph.successors()` etc. To allow algorithms to work with both classes easily, the directed versions of neighbors() and degree() are equivalent to successors() and the sum of in_degree() and out_degree() respectively even though that may feel inconsistent at times.

```
>>> DG=nx.DiGraph()
>>> DG.add_weighted_edges_from([(1,2,0.5), (3,1,0.75)])
>>> DG.out_degree(1,weighted=True)
0.5
>>> DG.degree(1,weighted=True)
1.25
>>> DG.successors(1)
[2]
>>> DG.neighbors(1)
[2]
```

Some algorithms work only for directed graphs and others are not well defined for directed graphs. Indeed the tendency to lump directed and undirected graphs together is dangerous. If you want to treat a directed graph as undirected for some measurement you should probably convert it using `Graph.to_undirected()` or with

```
>>> H= nx.Graph(G) # convert H to undirected graph
```

7 Multigraphs

NetworkX provides classes for graphs which allow multiple edges between any pair of nodes. The `MultiGraph` and `MultiDiGraph` classes allow you to add the same edge twice, possibly with different edge data. This can be powerful for some applications, but many algorithms are not well defined on such graphs. Shortest path is one example. Where results are well defined, e.g. `MultiGraph.degree()` we provide the function. Otherwise you should convert to a standard graph in a way that makes the measurement well defined.

```
>>> MG=nx.MultiGraph()
>>> MG.add_weighted_edges_from([(1,2,.5), (1,2,.75), (2,3,.5)])
>>> MG.degree(weighted=True, with_labels=True)
{1: 1.25, 2: 1.75, 3: 0.5}
>>> GG=nx.Graph()
>>> for n,nbrs in MG.adjacency_iter():
...    for nbr,edict in nbrs.iteritems():
...        minvalue=min(edict.values())
...        GG.add_edge(n,nbr,minvalue)


>>> nx.shortest_path(GG,1,3)
[1, 2, 3]
```

8 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=nx.petersen_graph()
>>> tutte=nx.tutte_graph()
>>> maze=nx.sedgewick_maze_graph()
>>> tet=nx.tetrahedral_graph()
```

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

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

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

```
>>> er=nx.erdos_renyi_graph(100,0.15)
>>> ws=nx.watts_strogatz_graph(30,3,0.1)
>>> ba=nx.barabasi_albert_graph(100,5)
>>> red=nx.random_lobster(100,0.9,0.9)
```

1. Reading a graph stored in a file using common graph formats, such as edge lists, adjacency lists, GML, GraphML, pickle, LEDA and others.

```
>>> nx.write_gml(red,"path.to.file")
>>> mygraph=nx.read_gml("path.to.file")
```

Details on graph formats: `/reference/readwrite`

Details on graph generator functions: `/reference/generators`

9 Analyzing graphs

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

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


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


>>> nx.clustering(G)
[0.0, 0.0, 0.0, 0.0]
```

With no nodes specified, functions that return Node Properties will return a list of values in an arbitrary order determined by the internal Python dictionary structure of the graph (which is returned by `Graph.nodes()` though it can change if the dictionary is resized).

The keyword argument with_labels=True returns a dict keyed by nodes to the node values.

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

Functions that return node properties, e.g. `Graph.degree()`, `clustering()`, etc, can For values of specific nodes, you can provide a single node or an nbunch of nodes as argument. If a single node is specified, then a single value is returned. If an nbunch is specified, then the function will return a list of values.

```
>>> nx.degree(G,1)
2
>>> G.degree(1)
2
>>> sorted(G.degree([1,2]))
[1, 2]
>>> sorted(G.degree())
[0, 1, 1, 2]
>>> G.degree([1,2],with_labels=True)
{1: 2, 2: 1}
```

Details on graph algorithms supported: `/reference/algorithms`

10 Drawing graphs

NetworkX is not primarily a graph drawing package but basic drawing with Matplotlib as well as an interface to use the open source Graphviz software package are included. These are part of the networkx.drawing package and will be imported if possible. See `/reference/drawing` for details.

First import Matplotlib's plot interface (pylab works too)

```
>>> import matplotlib.pyplot as plt
```

You may find it useful to interactively test code using "ipython -pylab", which combines the power of ipython and matplotlib and provides a convenient interactive mode.

To test if the import of networkx.drawing was successful draw G using one of

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

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

```
>>> plt.show()
```

command if you are not using matplotlib in interactive mode: (See Matplotlib FAQ)

To save drawings to a file, use, for example

```
>>> nx.draw(G)
>>> plt.savefig("path.png")
```

writes to the file “path.png” in the local directory. If Graphviz and PyGraphviz, or pydot, are available on your system, you can also use

```
>>> nx.draw_graphviz(G)
>>> nx.write_dot(G,'file.dot')
```

Details on drawing graphs: `/reference/drawing`