Analysis of Large-Scale Networks

NetworkX

JP Onnela

Department of Biostatistics
Harvard School of Public Health

July 17, 2014

Outline

What is NetworkX?

- NetworkX is a Python package for creating and manipulating graphs and networks
- The package, tutorial, and documentation are available at `http://networkx.lanl.gov`
- All examples were tested on Python 2.7.2 and NetworkX 1.6

How to Install NetworkX?

Approach A:

- The easiest option is to install Python, NetworkX, and several other modules using the one-click distribution by Enthought available at `www.enthought.com`
- The Enthought distribution is free for academic use

Approach B:

- One can first install install a program called `easy_install` and then use that to install NetworkX
- Download `ez_setup.py`
- Run it using `python ez_setup.py` or `sudo python ez_setup.py`
- Download a NetworkX Python egg, e.g., `networkx-1.5rc1-py2.6.egg`
- Run the Python egg using `easy_install networkx-1.5rc1-py2.6.egg`
- Python egg? A way of distributing Python packages

Nodes and Edges

- We first need to import the NetworkX module:

```
import networkx as nx
```

- We can create an instance of an undirected graph object:

```
G = nx.Graph()
```

- Adding nodes (one or several at a time):

```
G.add_node(1)
G.add_nodes_from([2,3])
G.add_nodes_from(['Tim', 'Tom'])
```

- Adding edges (one or several at a time):

```
G.add_edge(1,2)
G.add_edge('Tim', 'Tom')
G.add_edges_from([(1,2),(1,3)])
```

Nodes and Edges

- List of all nodes (list of objects):

```
G.nodes()
type(_)
```

- List of all edges (list of tuples of objects):

```
G.edges()
type(_)
```

- Number of nodes:

```
G.number_of_nodes()
```

- Number of edges:

```
G.number_of_edges()
```

Nodes and Edges

- We can also remove nodes and edges using similar methods:

```
G.remove_node(1)
G.remove_nodes_from([1,2])
G.remove_edge(1,2)
G.remove_edges_from([(1,2), (2,3)])
```

- To check for the existence of certain nodes or edges (returns True or False):

```
G.has_node(1)
G.has_edge(1,2)
```

- We can remove all nodes and all edges (the object G itself still remains):

```
G.clear()
```

Nodes and Edges

- Let's use one of NetworkX's network models to work with a slightly larger graph:

```
N = 100       # number of nodes
k = 6         # number of nearest neighbors
p = 0.05      # edge rewiring probability
G = nx.watts_strogatz_graph(N, k, p)
```

- List of neighbors of a given node:

```
G.neighbors(1)
```

- There is another more direct and faster way to find the neighbors of a node
- Note: Use this only to read the dictionary; **do not modify the dictionary directly**

```
G[1]
```

```
{0:  {}, 99:  {}, 4:  {}, 98:  {}, 3:  {}, 30:  {}}
```

- Degree by definition is the number of neighbors a given node has
- Since the `neighbors()` method returns a list, we can use its length to get degree:

```
len(G.neighbors(1))
```

- But more simply we can just use

```
G.degree(1)
```

Nodes and Edges

- We can use the `degree()` method to find the degrees of all nodes:

```
G.degree()
```

- This returns a dictionary with node-degree as key-value pairs
- Quiz: How do we get just a list of node degrees?

Nodes and Edges

- We can use the `degree()` method to find the degrees of all nodes:

```
G.degree()
```

- This returns a dictionary with node-degree as key-value pairs
- Quiz: How do we get just a list of node degrees?

```
G.degree().values()
sorted(G.degree().values())
```

- The latter makes use of the built-in `sorted()` function to create a new list that has the degrees ordered from low to high

Nodes and Edges

- **Exercise 1**: Write a script that loops through each node and prints out node ID and node degree

```
0 6
...
```

- **Exercise 2**: Write a script that loops through each node and prints out node ID and a list of neighbor node IDs

```
0 [1, 2, 3, 97, 12, 98]
...
```

- **Exercise 3**: Write a script that loops through each node and prints out node ID, node degree, and its average nearest neighbor degree

```
0 6 6.16666666667
...
```

Nodes and Edges

- **Exercise 1**: Write a script that loops through each node and prints out node ID and node degree

Nodes and Edges

- **Exercise 1**: Write a script that loops through each node and prints out node ID and node degree

```
for node in G.nodes():
    print node, G.degree(node)
```

- **Exercise 2**: Write a script that loops through each node and prints out node ID and a list of neighbor node IDs

- **Exercise 2**: Write a script that loops through each node and prints out node ID and a list of neighbor node IDs

```
for node in G.nodes():
    print node, G.neighbors(node)
```

- **Exercise 3**: Write a script that loops through each node and prints out node ID, node degree, and its average nearest neighbor degree

- **Exercise 3**: Write a script that loops through each node and prints out node ID, node degree, and its average nearest neighbor degree

```
for node in G.nodes():
    cumdeg = 0
    for neighbor in G.neighbors(node):
        cumdeg += G.degree(neighbor)
    print node, G.degree(node), float(cumdeg) / G.degree(node)
```

- **Exercise 3**: Write a script that loops through each node and prints out node ID, node degree, and its average nearest neighbor degree

```
for node in G.nodes():
    cumdeg = 0
    for neighbor in G.neighbors(node):
        cumdeg += G.degree(neighbor)
    print node, G.degree(node), float(cumdeg) / G.degree(node)
```

```
for node in G.nodes():
    if G.degree(node) > 0:
        cumdeg = 0
        for neighbor in G.neighbors(node):
            cumdeg += G.degree(neighbor)
        print node, G.degree(node), float(cumdeg) / G.degree(node)
```

- The first piece of code results in division by zero for isolated (degree zero) nodes, which leads to a run-time error

Basic Network Properties

- NetworkX provides a number of methods for computing network properties
- Note that the following are methods of the NetworkX module, not of graph objects
- **Clustering coefficient** characterizes the connectedness of a node's neighbors:

$$c_i = \frac{2t_i}{k_i(k_i - 1)}$$

- Here t_i is the number of connections among the neighbors of node i, and k_i is the degree of node i

```
nx.clustering(G)
nx.clustering(G,1)
nx.clustering(G,[1,2,3,4])
```

- Extract the **subgraph** induced by a set of nodes

```
g = nx.subgraph(G,[1,2,3,4,5,6])
g.number_of_edges()
```

- Extract **connected components** (returns a list of lists of the nodes in connected components; the list is ordered from largest connected component to smallest)
- Note that the method works for undirected graphs only

```
nx.connected_components(G)
```

- Sometimes we are interested in the **minimum spanning tree** of a graph or network (weighted or unweighted)

```
G = nx.watts_strogatz_graph(10000,6,0.2)
T = nx.minimum_spanning_tree(G)
T.number_of_nodes()
```

```
10000
```

```
T.number_of_edges()
```

```
9999
```

File Operations on NetworkX

- NetworkX has methods for reading and writing network files
- Two useful formats are edge lists and adjacency lists
- Column separator can be either a space (default), comma, or something else
- By default comment lines begin with the #-character

File Operations on Edge Lists

- Unweighted **edge list** is a text file with two columns: `source target`
- Weighted edge list is a text file with three columns: `source target data`
- Reading edgelists happens with the following commands:

```
G = nx.read_edgelist("edgelist.txt")
G = nx.read_edgelist("edgelist.txt", comments="#", delimiter=",", nodetype=int)
G = nx.read_weighted_edgelist("edgelist_w.txt", comments="#", delimiter=",",
    nodetype=int)
G.edges()
G.edges(data=True)
```

```
[(1, 2), (1, 3)]
[(1, 2, {’weight’: 5.0}), (1, 3, {’weight’: 5.0})]
```

- Edge lists can be written using the following commands:

```
nx.write_edgelist(nx.path_graph(4), "edgelist.txt", delimiter=’ ’)
```

```
0 1 {}
...
```

```
nx.write_edgelist(nx.path_graph(4), "edgelist.txt", delimiter=’ ’, data=False)
```

```
0 1
...
```

```
nx.write_weighted_edgelist(G, "edgelist_w.txt", delimiter=’ ’)
```

```
0 1 5.0
...
```

File Operations on Adjacency Lists

- Adjacency lists are typically unweighted and have the following format:
  `source target_1 target_2 target_3 ...`
- Reading adjacency lists:

```
G = nx.read_adjlist("adjlist.txt")
G = nx.read_adjlist("adjlist.txt", comments='#', delimiter=' ', nodetype=int)
```

- Writing adjacency lists can be done similarly:

```
G = nx.watts_strogatz_graph(100,6,0.2)
nx.write_adjlist(G,"adjlist.txt")
nx.write_adjlist(G,"adjlist.txt", delimiter=' ')
```

Generating Random Graphs

- NetworkX has some of the canonical random graphs readily implemented
- Erdős-Rényi (ER) random graph (one of the implementations):

```
G_er = nx.erdos_renyi_graph(1000, 0.15)
```

- Watts-Strogatz (WS) random graph:

```
G_ws = nx.watts_strogatz_graph(1000, 3, 0.1)
```

- Barabási-Albert (BA) random graph:

```
G_ba = nx.barabasi_albert_graph(1000, 5)
```

- Many others are available for creating, for example, Petersen graphs, Tutte graphs, etc.

Graph and Node attributes

- Attributes (weights, labels, colors, etc.) can be attached to graphs, nodes, and edges
- Graph attributes are useful if you need to deal with (are lucky enough to have) several graphs (e.g., in a longitudinal context)

```
G1 = nx.Graph(year=2004)
G2 = nx.Graph(year=2006)
G2.graph
```

```
{'year': 2006}
```

```
G2.graph['year'] = 2005
```

Graph and Node Attributes

- Node attributes can be used, for example, to represent demographic data (gender, etc.), or the status of a node in a dynamic process (susceptible, infectious, etc.)

```
G.add_node(1, sex = 'f')
G.add_node(2, sex = 'm')
G.add_nodes_from([3,4,5,7], sex = 'f')
```

- List the nodes with and without attributes:

```
G.nodes()
G.nodes(data=True)
```

```
[1, 2, 3, 4, 5, 7]
[(1, {'sex': 'f'}), (2, {'sex': 'm'}), (3, {'sex': 'f'}), ... ]
```

Graph and Node Attributes

- We can look up node attributes

```
G.node
G.node[1]
```

```
{1: {'sex': 'f'}, 2: {'sex': 'm'}, 3: {'sex': 'f'}, ... }
{'sex': 'f'}
```

- We can also modify node attributes

```
G.node[1]['status'] = 's'
G.node[2]['status'] = 'i'
```

- Make sure to keep these distinct:

```
G.nodes()           # method yielding a list of node IDs
G.nodes(data=True) # method yielding a list of node IDs with node attributes
G.node              # dictionary of node attributes
```

Edge Attributes

- Edge attributes can be used to represent edge-based data characterizing the interaction between a pair of nodes
- For example, in a communication network consisting of cell phone calls, we could use the number of phone calls made between the two nodes over a period of time (n) and the total duration of phone calls over a period of time (d) as edge weights

```
G.add_edge(1, 2, n = 12, d = 3125)
G.add_edge(1, 3, n = 9, d = 625)
```

- List all edges with and without data

```
G.edges(data=True)
G.edges()
```

```
[(1, 2, {'d': 3125, 'n': 12}), (1, 3, {'d': 625, 'n': 9})]
[(1, 2), (1, 3)]
```

Edge Attributes

- The short-cut syntax for accessing node attributes also works for edge attributes

```
G[1]
```

{2: {'d': 3125, 'n': 12}, 3: {'d': 625, 'n': 9}}

```
G[1][2]
```

{'d': 3125, 'n': 12}

- Adding weighted edges (one or several at a time):

```
# add one weighted edge
G.add_edge(1, 2, weight=1.3)
G.add_weighted_edges_from([(1, 2, 1.3)]) # results in "weight" as the key

# add multiple weighted edges with a common weight
G.add_edges_from([(1, 2), (2, 3)], weight=1.3)

# add multiple weighted edges with different weights
G.add_weighted_edges_from([(1, 2, 1.3), (1, 3, 4.1)])
```

[(1, 2, {'d': 3125, 'n': 12, 'weight': 1.3}), (1, 3, {'d': 625, 'n': 9, 'weight': 4.1}), (2, 3, {'weight': 1.3})]

Edge Attributes

- We now have an unweighted and weighted version (sometimes called node strength) of degree:

```
G.degree()
G.degree(weight="weight")
G.degree(weight="n")
G.degree(weight="d")
```

```
{1: 2, 2: 2, 3: 2}
{1: 5.3999999999999995, 2: 2.6, 3: 5.3999999999999995}
{1: 21, 2: 13, 3: 10}
{1: 3750, 2: 3126, 3: 626}
```

- Number of edges in the network:

```
G.number_of_edges()
G.size()
```

```
3
```

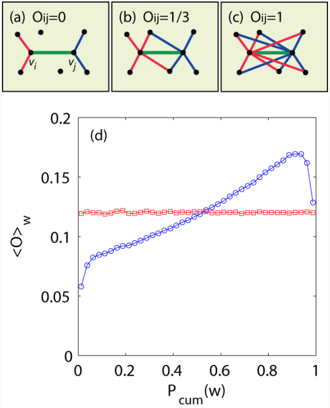
- Total weight of edges in the network:

```
G.size(weight="weight")
G.size(weight="n")
G.size(weight="d")
```

```
6.699999999999999
22.0
3751.0
```

Exercise: Edge Overlap

- We used the concept of edge overlap to examine the so-called weak tie hypothesis in “Structure and tie strengths in mobile communication networks” by J.-P. Onnela, J. Saramaki, J. Hyvonen, G. Szabo, D. Lazer, K. Kaski, J. Kertesz, and A.-L. Barabasi, PNAS 104, 7332 (2007)

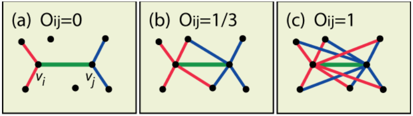


Exercise: Edge Overlap

- Write a function that computes the overlap O_{ij} of an edge (i, j), defined as the number of neighbors the nodes adjacent to the edge have in common, divided by the total number of neighbors the two nodes have combined:

$$O_{ij} = \frac{n_{ij}}{(k_i - 1) + (k_j - 1) - n_{ij}} \tag{1}$$

- Here k_i and k_j are the degrees of nodes i and j, respectively, and n_{ij} is the number of neighbors the two nodes have in common



- Hint: You will need to be able to find the degree of a given node, the neighbors of a given node, and the intersection of the neighbors of two nodes. Sets might be useful for this purpose.

Exercise: Edge Overlap

- Write a function that computes the overlap O_{ij} of an edge (i, j), defined as the number of neighbors the nodes adjacent to the edge have in common, divided by the total number of neighbors the two nodes have combined:

$$O_{ij} = \frac{n_{ij}}{(k_i - 1) + (k_j - 1) - n_{ij}}$$

```python
# overlap.py
# Function for computing edge overlap, defined for non-isolated edges.
# JP Onnela / May 24 2012

def overlap(H, edge):
    node_i = edge[0]
    node_j = edge[1]
    degree_i = H.degree(node_i)
    degree_j = H.degree(node_j)
    neigh_i = set(H.neighbors(node_i))
    neigh_j = set(H.neighbors(node_j))
    neigh_ij = neigh_i & neigh_j
    num_cn = len(neigh_ij)
    if degree_i > 1 or degree_j > 1:
        return float(num_cn) / (degree_i + degree_j - num_cn - 2)
    else:
        return None
```

- Let's examine the average overlap for a few Erdős-Rényi graphs

```
# overlap_test.py
# Explore edge overlap for ER graphs.
# JP Onnela / May 24 2012

import networkx as nx
import numpy as np
import matplotlib.pyplot as plt
from overlap import *

# Compute some overlap values for different ER networks.
ps = np.arange(0, 1.001, 0.05)
os = []
for p in ps:
    G = nx.erdos_renyi_graph(100, p)
    os.append(np.mean([overlap(G, edge) for edge in G.edges()]))

# Make a plot and save it to file.
fig = plt.figure(figsize=(10, 10))
line1 = plt.plot(ps, os, marker = "o", markersize = 10)
line2 = plt.plot(ps, ps)
plt.axis([0, 1, 0, 1])
plt.xlabel("Erdos-Renyi p-parameter", fontsize=15)
plt.ylabel("Average overlap <O>", fontsize=15)

#plt.show()
fig.savefig("../figs/overlap_test.pdf")
```

Exercise: Edge Overlap

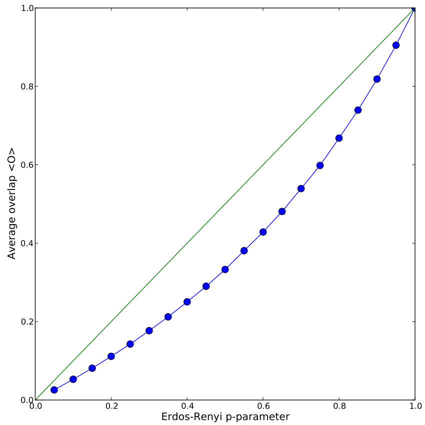


Figure : Average edge overlap $\langle O \rangle$ as a function of p for Erdős-Rényi networks. The figure was computed using $N = 1000$ for each network. The green line is the identity mapping and is shown for reference.

Exercise: Edge Overlap

- Consider an edge (i, j) in an Erdős-Rényi graph
- The probability for i to be connected to a node k, excluding node j is p; therefore the probability for both of them to be connected to node k is p^2
- Therefore the expected number of common neighbors is $(N-2)p^2$
- The expected (average) degree is given by $(N-1)p$
- This yields the following expression for edge overlap:

$$O_{ij} = \frac{(N-2)p^2}{2(N-1)p - 2 - (N-2)p^2}$$

- Taking the limit $N \to \infty$ this leads to

$$O_{ij} = \frac{Np^2}{2Np - 2 - Np^2} = \frac{p}{2 - 2/Np - p} = \frac{p}{2-p}$$

Exercise: Edge Overlap

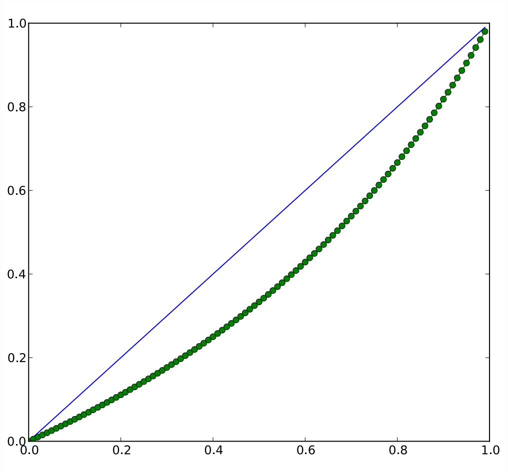


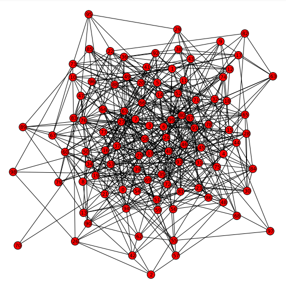
Figure : Plot of expected overlap $p/(2-p)$ in an Erdős-Rényi network as a function of p.

Network Visualization

- NetworkX uses the Matplotlib module for some simple network visualizations
- Let's start with an Erdős-Rényi network with $N = 100$ and $p = 0.11$
- Note that the actual drawing is deferred until the call to `show()`

```
import networkx as nx
import matplotlib.pyplot as plt

G = nx.erdos_renyi_graph(100,0.11)
plt.figure(figsize=(10,10))
nx.draw(G)
plt.axis("tight")
plt.show()
```



- We can print the figure directly to file by capturing the figure object and then using the `savefig` method
- Figure format will be specified by the file extension
- Possible to specify complete paths (absolute or relative)

```
import networkx as nx
import matplotlib.pyplot as plt

G = nx.erdos_renyi_graph(100,0.11)
fig = plt.figure(figsize=(10,10))
nx.draw(G, with_labels = False)
plt.axis("tight")
fig.savefig("../figs/vis2.pdf")
```

Network Visualization

- Sometimes we want to keep the node positions fixed
- It is possible to compute the node locations separately, resulting in a dictionary, which can then be used subsequently in plotting

```python
import networkx as nx
import matplotlib.pyplot as plt

G1 = nx.erdos_renyi_graph(100,0.01)
G2 = nx.erdos_renyi_graph(100,0.02)
G3 = nx.erdos_renyi_graph(100,0.04)
G4 = nx.erdos_renyi_graph(100,0.08)

fig = plt.figure(figsize=(10,10))
pos = nx.spring_layout(G4,iterations=500)
plt.subplot(2,2,1)
nx.draw(G1, pos, node_size=40, with_labels=False); plt.axis("tight")
plt.subplot(2,2,2)
nx.draw(G2, pos, node_size=40, with_labels=False); plt.axis("tight")
plt.subplot(2,2,3)
nx.draw(G3, pos, node_size=40, with_labels=False); plt.axis("tight")
plt.subplot(2,2,4)
nx.draw(G4, pos, node_size=40, with_labels=False); plt.axis("tight")
fig.savefig("../figs/vis3.pdf")
```

Network Visualization

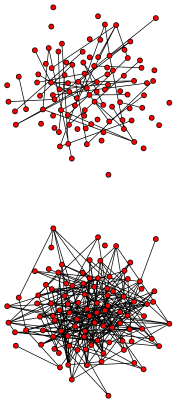


Figure : Erdős-Rényi networks with different values for the p-parameter using fixed node locations.