Week 10.2 - Social Network Analysis

Social network analysis (SNA) can be briefly described as the "study of human relationships using graph theory." The SNA discipline has been used in a number of areas to analyze people or collections of people through their relationships. [Review slides 1–4, 7 from Cheliotis.]

Python package for networks

We'll use the Python package networkx to build networks that we can analyze. This package can be installed with Anaconda. At the command prompt:

```
$  conda install networkx
```

We'll start with some network basics in a Python interpreter window.

```
>>> import networkx as nx    #  this allows us to use a shorter name for the package
```

Let's do some initial examples to build networks with the networkx package. A network consists of a number of nodes, which may be connected by edges.

Undirected Graphs

Create a graph called g. The Graph() constructor function is for undirected graphs.

```
>>> g = nx.Graph()
```

Nodes can be named with numbers or strings. If you add an edge and the node doesn't exist already, then it adds it to the graph. Or add_edge will connect an existing node.

```
>>> g.add_edge(1,2)
>>> g.add_node("spam")
>>> g.nodes()
[1, 2, 'spam']
>>> g.edges()
[(1, 2)]
>>> g.add_edge(1,"spam")
>>> g.edges()
[(1, 2), (1, 'spam')]
```

Note that edges() returns a list of tuples, where each tuple gives the node at the beginning and end of the edge. (This graph is an undirected graph, so it doesn't matter which node is the beginning of the edge and which is the end. They are both really ends.)

We can add lists of nodes and edges.

```
>>> g.add_nodes_from([3,4])
>>> g.add_edges_from([(1,3),(3,4)])
>>> g.nodes()
[1, 2, 3, 'spam', 4]
>>> g.edges()
[(1, 2), (1, 3), (1, 'spam'), (3, 4)]
```

You can attach attributes to the graph or to any node or edge. The attributes can be any Python type, typically strings and numbers. Each attribute has a keyword name.

We can attach an attribute with the keyword name 'color' to a new node as we add it, or we can use an indexing notation to add the attribute to an existing node. Indexing the graph with a node name returns a dictionary representation of the data of the node.

```
>>> g.add_node(5, color='blue')
>>> g.node[1]['color'] = 'red'          # g.node[1] is the dictionary of existing node 1
>>> g.nodes(data=True)                  #  lists each node with its data dictionary
[(1, {'color': 'red'}), (2, {}), (3, {}), (4, {}), (5, {'color': 'blue'}), ('spam', {})]
```

Edges can also have a data dictionary attached to them, which is accessed by giving a double index with the end nodes of the edge. One possible attribute has the special name 'weight', where it is special because some functions can use the 'weight' attribute.

Now let's attach an attribute with the name 'weight' to either a new edge or added to an existing edge, using an index notation for an existing edge.

```
>>> g.add_edge(2,5, weight=4.7)
>>> g.edge[1][2]['weight'] = 1.2      #g.edge[1][2] is dictionary of edge from 1 to 2
>>> g.edges(data=True)
[(1, 2, {'weight': 1.2}), (1, 3, {}), (1, 'spam', {}), (2, 5, {'weight': 4.7}), (3, 4, {})]
```

The neighbors function gives nodes connected to a node.

```
>>> g.neighbors(1)
[2, 3, 'spam']
```

The adjacency matrix gives a square matrix with the nodes as rows and columns, filled in by which nodes are connected by edges. Edges without weights are given 0 and 1, and with a weight, the weight value is given.

```
>>> adj_matrix = nx.adjacency_matrix(g)
>>> type(adj_matrix)
<class 'scipy.sparse.csr.csr_matrix'>
```

```
>>> amatrix = adj_matrix.todense()
>>> type(amatrix)
<class 'numpy.matrixlib.defmatrix.matrix'>
>>> amatrix
matrix([[0, 1, 1],
        [1, 0, 0],
        [1, 0, 0]], dtype=int64)          # this result for a small graph with 3 nodes
```

The number of edges connecting a node to the graph is known as the degree. The degree() function returns a dictionary of node names and the degree of the node. We can also find the node with the minimum or maximum degree.

```
>>> deg = nx.degree(g)
>>> type(deg)
<class 'dict'>
>>> deg
{1: 3, 2: 2, 3: 2, 4: 1, 5: 1, 6: 0, 'spam': 1}
>>> min(deg.values())
0
>>> max(deg.values())
3
```

Or we could sort the dictionary of degrees to get a list of items in order from the highest degree down to the lowest.

```
>>> sorteddeg = sorted(deg.items(), key = lambda x: x[1])
>>> for node in sorteddeg:
   print(node)
```

Directed Graphs

We can create a directed graph using the DiGraph() constructor function.

```
>>> dg = nx.DiGraph()
>>> dg.add_weighted_edges_from([(1,2,.5),(3,1,.75)])
```

Directed graphs have the same functions as undirected graphs, for example:

```
>>> dg.nodes()
[1, 2, 3]
>>> dg.edges(data=True)
[(1, 2, {'weight': 0.5}), (3, 1, {'weight': 0.75})]
```

But they also have additional functions, such as out_degree() to say how many edges are leaving a node or to give the weights of those edges because edges now have direction.

```
>>> dg.out_degree(1)
1
>>> dg.out_degree(1, weight='weight')
0.5
```

Another function on any graph is to find the connected components.

```
>>> nx.connected_components(g)
<generator object connected_components at 0x1006917e0>
>>> comp_list = list(nx.connected_components(g))
>>> comp_list
[{1, 2, 3, 4, 5, 'spam'}]
```

All the nodes are connected, so there is only one component on the list. Let's add another node, not connected to anything, and it will form a separate component.

```
>>> g.add_node(6)
>>> comp_list = list(nx.connected_components(g))
>>> comp_list
[{1, 2, 3, 4, 5, 'spam'}, {6}]
```

More information about network functions can be found at the package documentation http://networkx.lanl.gov/tutorial/tutorial.html. For example, networkx can also compute cliques.

If you're on a system where matplotlib.pyplot works, you can draw several shapes of the graph. And if you don't have matplotlib, use "conda install matplotlib".

```
>>> import matplotlib.pyplot as plt
>>> nx.draw(g)        # draw_random, draw_sphere, etc.
>>> plt.show()
```

For some of the SNA functions, there are lots of nodes with 1 degree or less, and we should get rid of those in order to look at the core nodes of the network.

```
>>> def trim_degrees(g, degree=1):
...     g2 = g.copy()
...     d = nx.degree(g2)
...     for n in g2.nodes():
...         if d[n]<=degree: g2.remove_node(n)
...     return g2
...
>>> core = trim_degrees(g)
>>> len(core)
3
>>> core.nodes()
[1, 2, 3]
```

```
>>> core.edges()
[(1, 2), (1, 3)]
```