Lab 1: Basic Network Properties (Newman, 6.1-6.8)

In this lab, we will introduce the main computational tool we will use in this course. `zen` is a Python library built to load, analyze, and manipulate networks. If you've never used Python, have no fear. It is simple and powerful. If you've done any MATLAB, C, Java programming, you will pick it up easily (tutorials, if you want one). It is arguably the most popular language for analytics used in industry, so it is a great thing to learn.

Installing

Below are the instructions for installing Python and `zen` (the tricky part is installing all the dependencies for `zen`). These instructions assume that you are *not* using an automatically curated Python distribution like Enthought's Canopy or Anaconda. If you are using these, then we can still install `zen` (provided you get the 32-bit version if you are using Windows). Come talk to me if this is the case.

Windows

1. Install the 32-bit version of Python 2.7.13. Actually any version <3.0 and >2.7.6 should work fine, but it is very important that the version is 32-bit (this has to do with the compiler we need to use on Windows). This installation will create the directory `C:\Python27`. If you already have Python installed, it is likely you (justifiably) chose the 64-bit version. Unfortunately, you must still install the 32-bit version. It is possible to have both on your computer. To check what version of Python you have, open the Windows command line (hit Windows-R, type `cmd`, and press enter). At the prompt type `python`. Somewhere in the lines that print out it will either say 64-bit or 32-bit.
2. Install the minGW compiler, which is a free C compiler available on Windows. From the "Basic Setup" section, select the "mingw32-base" package and then select "Apply Changes" from the "Installation" menu. This will create the directory `C:\MinGW`.
3. We now need to add these programs to the system path, so we can use them on the command line. Open the Control Panel and search for "path". Click on the link that says "Edit the system environment variables". Click on the button "Environment Variables" and in the System Variables list (the bottom panel), find the one called "Path". Select it and click "Edit...". Add the following to the end of the path (a semicolon separates different entries) for `python` itself, `pip` (which you will use to install other python libraries), and the MinGW compiler:

   ```
   ;C:\Python27;C:\Python27\Scripts;C:\MinGW\bin
   ```
4. Open Notepad and create a file containing the following two lines. Save it as `C:\Python27\Lib\distutils\distutils.cfg`. When you save, make sure to select "All files (*.*)" from the Save as type dropdown list. Otherwise it will save it as a .txt file.

   ```
   [build]
   compiler=mingw32
   ```

   This tells Python which compiler to use to compile `zen`.
5. Before we install `zen`, you will need a few dependencies (other libraries `zen` depends on). We will install the following packages: numpy (MATLAB-like mathematical package), scipy (additional mathematical package), matplotlib (a library for plotting figures), networkx (another library for network analysis), and cython (a library for compiling Python to C code). While these can be compiled directly, this is tricky. Binaries are available and much easier to install. Download this zip file (originally from this website). Unzip the contents to a temporary directory. On the Windows command line navigate to this directory. The easiest way to do this is to copy the full path of where you unzipped the files, then type on the command line `cd` followed by pasting the full path, for example: `cd C:\Users\Username\Downloads` (or you can do relative navigation by typing `cd foldername` to navigate into the directory `foldername`. Tab completion allows you to start typing the name and then hitting the tab key to automatically complete the rest of the folder/file name. To go up a directory type `cd ..`). Once you are in the right directory, enter these lines separately (hitting enter between commands):

   ```
   pip install numpy<tab complete the rest>
   pip install scipy<tab complete the rest>
   pip install matplotlib<tab complete the rest>
   pip install networkx<tab complete the rest>
   pip install Cython<tab complete the rest>
   ```
6. Download the `zen` source as a zip file. Unzip this to a temporary location. Using the Windows command line navigate to the `zenlib-master\src` directory. Once sitting inside the `zenlib-master\src` directory, run the following command. This will compile and install zen. It may take some time and print out a lot of text in the meantime.

```
python setup.py install
```

7. Close this command line window and open a new one. Type `python` to start Python in "interactive mode". Now that you're in Python, type `import zen` and hit enter. If it doesn't complain, then you're all set up! If you get an error, come find me.
8. The last step is to build the documentation for `zen`. Install the epydoc program. On the command line run the following command (in the directory where you want the documentation to be). It will create a folder called `zen-pydoc`.

```
python C:\Python27\Scripts\epydoc.py --html -o zen-pydoc zen
```

OSX

1. Mac systems are derived from unix, so they already have Python installed by default. You may be using this version or you may have installed another one along the way. To determine what you are using, open Terminal (press Cmd+Space and type `terminal`, then hit enter) and type `which python` (then hit enter). If the output is `/usr/bin/python` then you are using the default Python installation. The following step should work no matter which version you are using.
2. Check now to see if you have `pip` installed, by typing `pip` in Terminal and hitting enter. If it says something like `-bash: pip: command not found`, then you do not have it installed. To install it run: `sudo easy_install pip` (using `sudo` requires you to enter the password for your user on your computer – it is like doing something in administrator mode).
3. In Terminal run the following commands one at a time (some of these may already be installed, so if it says "Requirement already satisfied", just move on to the next one):

```
sudo pip install numpy
sudo pip install scipy
sudo pip install matplotlib
sudo pip install networkx
sudo pip install cython
sudo pip install epydoc
```

4. Download the `zen` source as a zip file. Unzip this to a temporary location. Within Terminal navigate to the `zenlib-master\src` directory. Relative navigation is done by typing `cd foldername` to navigate into the directory `foldername` (tab completion allows you to start typing the name and then hitting the tab key to automatically complete the rest of the folder name). To go up a directory type `cd ..`
5. Once sitting inside the `zenlib-master\src` directory, run the following command. This will compile and install zen. It may take some time and print out a lot of text in the meantime.

```
sudo python setup.py install
```

6. Close this terminal window and open a new one. Type `python` to start Python in "interactive mode". Now that you're in Python, type `import zen` and hit enter. If it doesn't complain, then you're all set up! If you get an error, come find me.
7. The last step is to build the documentation for `zen`. In Terminal run the following command (in the directory where you want the documentation to be). It will create a folder called `zen-pydoc`.

```
epydoc --html -o zen-pydoc zen
```

Documentation Organization

Go inside the `zen-pydoc` folder you just created and double click on `index.html`. From here you can browse to find the objects and functions you can use to work with networks. We'll introduce it a bit at a time. When you browse the documentation, typically you start in the top left frame and select the submodule you want to look at. Then you use the other frames to navigate deeper. As we move along, whenever you see a new function, make sure to look it up in the `zen` documentation!

Running Your Python Code

In a few moments you will create a .py file – in this case called `lab1.py`. To run this Python script, you:

1. create/edit the file in a text editor of some kind (nearly anything will work, but there are some that do syntax highlighting)
2. save it in some location,
3. in either the Windows command line or Terminal navigate to the directory that contains your script, and

4. type `python lab1.py` and hit enter. When you make changes to your .py file (don't forget to save!), you can simply run the same command to run the updated script. Both command line and Terminal store your history of commands, which you can retrieve using the up and down arrows. So, the fast way is to rerun your script is to press the up arrow and then enter.

Begin

Create a `lab1.py` file to use for the following exercises. At the top of the file create a commented out section (using `#`) and place your names and UTD NetIDs there. Put all code/analysis for a given lab into only one file – this makes it much easier to grade. Please make sure you label your print statements, so we know what you are printing. This lab asks a lot of questions, many of them rather minor – make sure to answer all the questions that I write out in the text. Some may seem silly/trivial, but they are put there to make you think.

The following command imports the `zen` library so you can use it. Place this at the top of your file.

```
import zen
```

Sections 6.1 - 6.2: Networks

In `zen`, build the small network in Figure 6.1a. Let's learn how to do this - it's actually quite simple. Begin by creating a new `Graph` object:

```
G = zen.Graph()
```

Look up `Graph` in the `zen` documentation. Do this by navigating using the upper left panel to find `zen.graph` - click on this and then click on the `Graph` object link in the right panel that comes up. You'll see that it represents an undirected graph - exactly what you need! This object has quite a few built-in methods that help us manipulate and study networks. Let's use the `add_edge(u,v)` function to build up the network:

```
G = zen.Graph()
G.add_edge(1,2)
G.add_edge(2,3)
...
```

Continue adding edges until you have formed the entire graph. Notice that the numbers 1, 2, 3, … are now the names of the nodes and the nodes get added automatically if they don't already exist. The second line `G.add_edge(2,3)`, for example, only adds one new node (node 3) since node 2 was created in the previous line. In this case, it doesn't matter which order you put the nodes in the function call.

An important distinction that `zen` uses is the notion of node names (objects) versus node indices. Node indices are always integers and the first node is given the index of 0 and the rest count up from there as they are added. If no nodes have ever been removed, then the indices count from 0 up to the number of nodes (removing nodes leaves holes in the indexing - but you can call `compact()` to reindex them). The names (node objects) are unique identifiers that do not have to be numbers. We will use node objects to hold the names of people, characters, locations, etc when we study real networks (because node 14 doesn't really give us much intuition about what that node is).

Let's look more carefully at the documentation again. You'll notice that `Graph` has two functions that look very similar: `add_edge(u,v)` and `add_edge_(u,v)`. You'll see this pattern all over `zen`. Functions that end in an underscore take node *indices* as input and functions not ending with an underscore take node *objects* as input. So in our current code, we are passing the node objects even though they are numbers. The node indices are automatically being generated. Generally we don't care so much what the node indices are - they are useful tools so that we can order the nodes in a systematic way, but usually we don't care whether a particular node is indexed as 5 or as 38.

Now that the network is complete, check that `G.num_nodes` and `G.num_edges` match your expectations (note that these are *properties* of the Python `Graph` object, not functions, which is why they don't have parentheses). You could do this with separate print statements, but that gets confusing as soon as you are printing out a lot of stuff, so try this format:

```
print '#Nodes: %i, #Edges: %i' % (G.num_nodes,G.num_edges)
```

(Be careful copying an pasting these code lines. Sometimes the characters get copied over strangely and python complains. I would recommend typing it out yourself!)

Nearly all programming languages have this sort of string formatting. You create a string and insert "plugs" into which you can drop the value of a variable. These "plugs" start with `%`. The most useful are `%i` (used for integers), `%1.2f` (used for decimals - the 2 indicates we want two digits after the decimal point), and `%s` (used for strings). Following the string itself, you place another `%` and then the tuple of variables that you plug into the string, in the order they appear.

Ensure that there exists a link between nodes 3 and 4 by checking that `G.has_edge(3,4)` is true. Similarly check that there is no edge between nodes 4 and 6 by checking that `G.has_edge(4,6)` is false. Again we want to label the output nicely, so do something like this:

```
print 'Has 3-4 edge: %s' % G.has_edge(3,4)
```

Notice that if you only insert one variable into a string, then it does not need to be in a tuple. Also, even though the function returns a boolean (`True` or `False`) it is automatically cast into a string.

Print out the corresponding adjacency matrix (look up the `matrix()` method of the `Graph` object). Find the 1 and 0 in the adjacency matrix that correspond to the edge between nodes 3 and 4 and the lack of an edge between nodes 4 and 6, respectively.

Note! At this point it is possible that your adjacency matrix doesn't match your expectations. The matrix is displayed by node *index*, so depending on how you added the edges, you might be naming a node as "4", but its node index is not 4! To get around this problem you can first add the nodes directly using the `G.add_node()` function. If you're tired of typing this many times, you can wrap it in a for loop to make it quick:

```
for i in range(1,7):
    G.add_node(i)
```

Check that the adjacency matrix of an undirected network is symmetric (do this first visually) and also computationally by running the command `A == A.transpose()`, provided that `A` is the adjacency matrix. Note that `transpose()` is a function given to `A` because `A` is a NumPy array.

Section 6.3: Weighted Networks

The graph in Figure 6.1b has multiple edges between the same nodes and is called a multigraph. Although we won't use this representation very much (because it complicates things), it is a more realistic model of real life - rarely are interactions on different topics all the same. For example, you might trust someone very differently as a work colleague than as a personal friend. For now, as the book mentions, interpret each edge in the figure as representing unit weight - so a pair of nodes with 2 edges between them creates a condensed representation with a single edge of weight 2. Construct this weighted network in `zen` and print the corresponding adjacency matrix. Entering edges with a specified weight in `zen` only requires passing an additional parameter: `G.add_edge(2,3,weight=2)`. Remember if you used `G` in the section above, you will need to either reinitialize the variable (`G = zen.Graph()`) or create a new one with a different name (e.g., `G2`) for this section.

Section 6.4: Directed Networks

In `zen`, build the small directed network in Figure 6.2 and compare the adjacency matrix to the one in the textbook. In this case you will use `zen.DiGraph` object, which represents a directed network. When you use `G.add_edge(u,v)` in a `DiGraph`, you add the directed edge from `u` to `v` - so the order matters.

Note that a directed network no longer has a symmetric adjacency matrix. The textbook defines the adjacency matrix such that the entry $A_{ij}=1$ if there is an edge from node `j` to node `i`. However, other sources define it the opposite way - in `zen` $A_{ij}=1$ if there is an edge from `i`

to `j`. Admittedly, this can lead to some confusion, but the important thing is to be consistent. Just keep in mind that where the textbook uses `A`, we will use `A.transpose()` in `zen`; when the text uses A^T, use `A` in `zen`. In your printout (using the `zen` convention), the edge from node 6 to node 4 makes $A_{64}=1$, which is the sixth row and the fourth column. Notice that the $A_{46}=1$ entry is zero because there is no edge from 4 to 6.

There are a number of interesting things we can do with directed graphs. The next few sections take a look at several of them, however, we will see more as our tools become more advanced.

Section 6.4.1: Cocitation & Bibliographic Coupling

This section explores two complementary ways of meaningfully changing a directed graph into an undirected graph. Although they are originally motivated by citation networks - networks which encode which papers cited other papers as references - these tools can be used in a variety of applications.

An edge with weight `k` between nodes in the cocitation network indicates that these nodes were both pointed to (cited) by `k` other nodes. An edge with weight `k` between nodes in the bibliographic coupling network indicates that these nodes both point to (cite) `k` of the same other nodes.

Let's take a look at how this works for a dataset about Linear Algebra. ProofWiki is a website that has mathematical definitions and proofs. Because they are laid out in wikipedia format, it is relatively easy to scrape the pages and create a network. In this case, let's look at a network which includes all the mathematical definitions on the ProofWiki website. An edge points from node a to node b if definition a has node b in it's description. For example, the article on Linear Span has the text: "The *linear span* can be interpreted as the set of all *linear combinations* (of *finite* length) of these *vectors*." This means that the Linear Span has edges out to Linear Combination, Finite, and Vector. In this case, to make it smaller and faster, I've limited it to the definitions that are contained within the topic of Linear Algebra. Load this network (to do this, make sure you download the `proofwikidefs_la.gml` file into the same directory as your python file):

```
G = zen.io.gml.read('proofwikidefs_la.gml',weight_fxn = lambda x: x['weight'])
```

We can use a variety of formats, but the GML format is a useful one to keep the full names of the nodes if the node names have spaces in them. The last part is some "advanced" Python which defines a function in place. It is describing how the value of the edge weights should be read in. In this case, there is a property in the gml called "weight" (sometimes it is called something different, e.g., value).

Create a function to compute the new cocitation network (and then ultimately the cocitation matrix), assuming an original network that is weighted (the text describes how to incorporate weights into the cocitation calculation). The simple way is to weight the cocitation network using just the count of common in-bound neighbors. You may want to do this first and then develop the weighted extension. You will want to look at the functions `G.in_neighbors()` and `G.weight()`. One subtlety to be aware of: in `zen`, if you add an edge with zero weight, it still is considered an edge, so it will be used to find (in/out)-neighbors of nodes. Later in the course, we'll see some algorithms on graphs in which defining zero weights can be useful. This is a case where the graph has additional information beyond what the adjacency matrix contains. Here, we don't want zero weight edges, so be careful not to add them!

```
def cocitation(G):
   write algorithm here
   ...

   return G_cocitation
```

You'll need to create a new `Graph` object and then build it up according to the rules of the cocitation graph. Then use this function to calculate the cocitation matrix of the ProofWiki network. Compare the matrix of this cocitation network with the formula in the book $C=AA^T$ (recall that you need to switch `A` to `A.transpose()`). In NumPy (one of the math modules in Python) to calculate a matrix multiplication, you call the function `numpy.dot(A,B)`

```
C1 = numpy.dot(A.transpose(),A)
C2 = G_cocitation.matrix()
Cdiff = C1-C2
print 'Difference between cocitation methods: %i' % Cdiff.sum().sum()
```

Because we're now using the NumPy module, you need to place `import numpy` up at the beginning of the file. In the last line we are essentially summing up all the entries in the matrix `Cdiff`, by summing along one direction to get a vector of sums, then summing this vector.

This sum should be zero - but it isn't! Why not? What did we miss? Fix the above lines to make sure that you account for this detail. Incidentally, in order to do this, it might be helpful to know how to index through a matrix. It works just like a list, but takes two indices: `A[i,j]` is the A_{ij} entry.

Now that you've got that sorted out, let's look at what we have created. In particular, let's look at the definition "Linear Combination" (this is the node name). Print out the neighbors of this node in the cocitation network along with the weights of the edges they share. Write a concise phrase that captures the meaning of these neighbors. Compare these with the in-neighbors of "Linear Combination" in the original graph. Which set of neighbors is more specific and which is more general? How does this connect with what you know about cocitation?

If you have extra time, you can try implementing the bibliographic coupling method as well, but it isn't required.

Concept question: What is the original directed network (draw or describe) that has a cocitation matrix given by:

$$\begin{pmatrix} 0 & 4 & 4 & 4 & 4 & 4 \\ 4 & 0 & 4 & 4 & 4 & 4 \\ 4 & 4 & 0 & 4 & 4 & 4 \\ 4 & 4 & 4 & 0 & 4 & 4 \\ 4 & 4 & 4 & 4 & 0 & 4 \\ 4 & 4 & 4 & 4 & 4 & 0 \end{pmatrix}$$

What is the corresponding bibliographic coupling matrix for this network?

Concept question: Is it possible that two different (potentially weighted) original graphs G_1 and G_2 have the same cocitation and same bibliographic coupling graphs (e.g., $C_1=C_2$ and $B_1=B_2$)? If so, give an example.

Section 6.4.2: Acyclic Networks

Implement the simple algorithm introduced in this section (as a new function) to determine whether a network is acyclic or not. Run your algorithm on the three networks supplied: `acyclic1.edgelist`, `acyclic2.edgelist`, and `acyclic3.edgelist`. The edgelist format is a simple format to save a network in which each line has the name of one node, a space (or tab or some sort of whitespace), and then the name of another node – each of these lines specifies an edge between the two nodes (it can be interpreted as directed or undirected way, so we need to specify). To read in this kind of network use the `zen.io.edgelist` module:

```
G = zen.io.edgelist.read('acyclic1.edgelist',directed=True)
```

Since you'll reuse this code (at least three times), it makes sense to define a small function to run the algorithm. Then beneath this function call it for each of the three graphs.

You'll probably want to use a `while` loop. Also, a very relevant (and convenient) implementation detail of `zen` is that when you remove a node, then all the edges to/from that node are also automatically removed.

Section 6.5: Hypergraphs

Know and understand the concept of a hypergraph. There is more to know, but we won't go into it during this course.

Section 6.6: Bipartite Networks

You're probably figuring out some of the reoccurring patterns of network science already. Here we introduce a new type of network, the bipartite network, and one of the first things we would like be able to do is take the bipartite (or two-mode) network and transform it into an undirected network (the one-mode projection). We do this not only because the undirected graph is the "common denominator" of graphs, but also because the resulting graph can be quite informative! The book describes doing this in two ways focusing on either the "participants" or the "groups".

In this exercise, we'll take a look at a portion of the Internet Movie DataBase (IMDB). The original database had a list of all actors and actresses and which movies they had been in. However, this was a little too much data, so I found a list of the top actors and actresses of 2013 and only kept those actors and actresses. I took this list and built a bipartite graph with actors and actresses as one type of node and movies as another type of node. Let's load this into `zen`:

```
B = zen.io.gml.read('2013-actor-movie-bipartite.gml')
```

The network you have just read in is a bipartite network - so also look up `BipartiteGraph` in `zen`. You'll see that this type of graph has some unique characteristics. For example, it has a built in way of dealing with the two types of nodes - one group has "U" nodes and the other group has "V" nodes. In the network you've just loaded, the actors and actresses are U nodes and the movies are V nodes.

Your task is to write the code to create the weighted one-mode projection of this bipartite graph. Since there are two such projections, let's choose the one which yields an actor/actress network. It isn't a bad idea to create a few simple test networks (ones that you know the right answer) to make sure you implemented everything correctly. This implementation should feel quite similar to the cocitation function you wrote above.

Let's look at this network a little more closely. Consider the actors: Will Ferrell and Jason Statham. Print out their immediate neighbors, e.g.,

```
print 'WILL FERRELL: %s' % G.neighbors('Will Ferrell')
```

(this works because the node objects are the actor/actress names). Suppose you don't know who Will Ferrell and Jason Statham are - how could you use this information about their neighbors in the one-mode projection to learn more about them?

It is useful to reflect on how our biases in sampling the data are reflected in the network. Recall this is a sample of the much larger actor-movie network and was chosen according to the top (Hollywood) actors and actresses in 2013. What are the number of neighbors of new-comers like Zac Efron and old-timers like Clint Eastwood? Is it surprising that they have low degrees in this network?

Section 6.7: Trees

We've already looked at acyclic networks. Describe how a directed tree is different from an acyclic network (recall, a directed tree is a directed network whose underlying undirected graph is a tree). Draw a 7 node directed acyclic graph that is not a directed tree, then highlight which edges you would need to remove to make the graph a directed tree. Draw this graph so that all edges are pointing downwards or sideways. Is this set of edges always unique? (don't try to draw this in your code - draw it in your attached PDF).

Section 6.8: Planar Networks

You may have learned that on a map you only ever need four colors if you want to make sure that no country/state touches another country/state of the same color. This idea is explained in terms of graph theory in this section. This is due to the fact that the network of countries is a planar network.