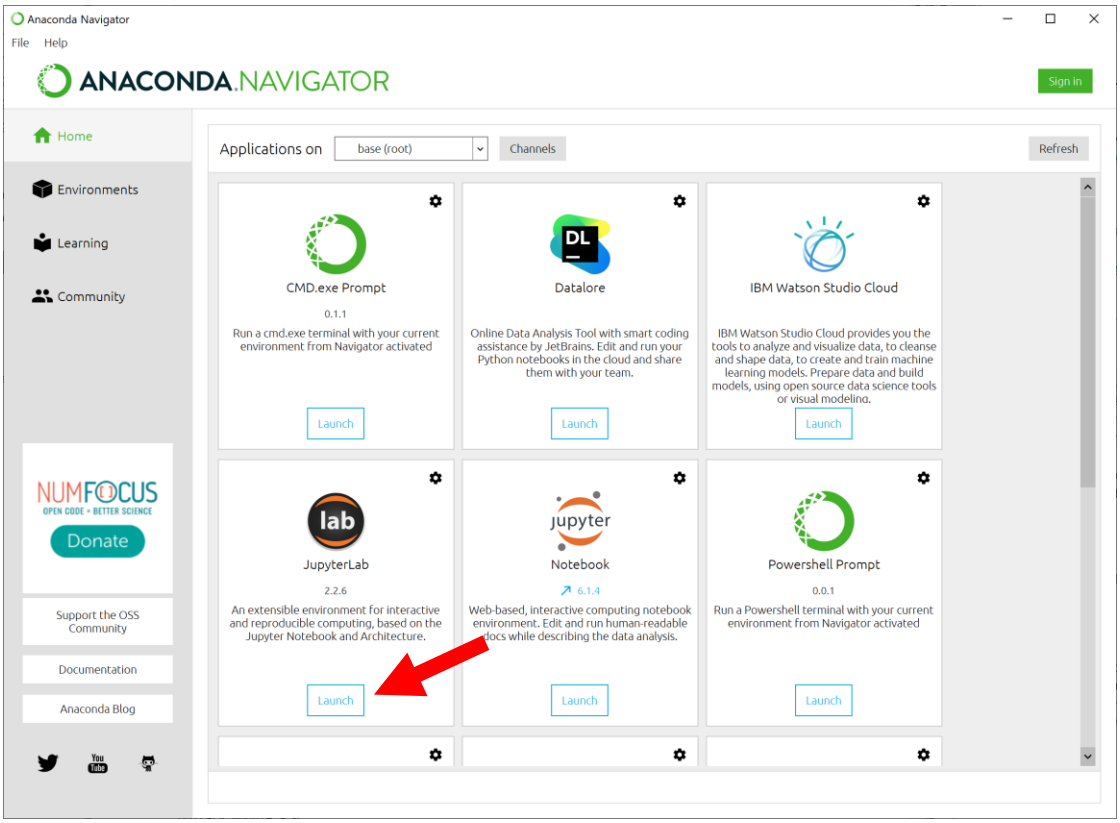
Lab 1: Basic Network Properties (Newman 6.1-6.8)

In this lab, we will introduce the main computational tool we will use in this course. networkx 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

Install Anaconda. This will install Python. In this course we will make use of the JupyterLab development environment, but there are many, many ways to run Python code. I will show you some of these to help demystify what is actually happening in the JupyterLab interface. You can launch JupyterLab from the Anaconda Navigator.



Begin

Create a `lab1_abc123456.ipynb` file to use for the following exercises. At the top of the file create a markdown section (there is a dropdown box in the toolbar that allows you to select "Code" or "Markdown") and place your name(s) and UTD NetID(s) 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 `networkx` library so you can use it in your code. It renames the library as `nx` simply so there is less writing involved when you use it. Place this at the beginning of your file.

```
import networkx as nx
```

To run a block of code you can either use the toolbar icon, or can hit Shift+Enter.

Sections 6.1 - 6.2: Networks

In `networkx`, 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 = nx.Graph()
```

Take a look at the `Graph` object in the reference documentation by clicking the link above. Soon you will feel comfortable navigating and finding documentation on your own, but I will provide links when I introduce major new functions and classes. You'll see that it represents an undirected graph - exactly what you need! The first section of the documentation provides an overview of how to use the `Graph` object and lower down is a complete list of methods (functions) that can be used to manipulate the `Graph` object. Let's use the `add_edge(u,v)` function to build up the network:

```
G = nx.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, since it is an undirected graph, it doesn't matter which order you put the nodes in the function call. The names are unique identifiers that do not have to be numbers. We will use node names 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).

Now that the network is complete, check that `G.number_of_nodes()` and `G.number_of_edges()` match your expectations (look these up in the `Graph` documentation you already had open). 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.number_of_nodes(),G.number_of_edges()))
```

(Be careful copying and 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 `adjacency_matrix()` function). Note that this function returns a *sparse* matrix, which only keeps track of the nonzero elements of the matrix. To see the full matrix we can request the *dense* version:

```
print(nx.adjacency_matrix(G).todense())
```

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 the order in which nodes were added, so depending on how you added the edges, you might be naming a node as "4", but its index in the adjacency matrix is not the 4th row/column! To get around this problem you can first add the nodes directly using the `G.add_nodes_from()` function,

```
G.add_nodes_from(range(1,7))
```

where the command `range(1,7)` generates the list `[1,2,3,4,5,6]`. The `add_nodes_from()` function is a shortcut to calling the `add_node()` function multiple times. 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. The shortcut property `A.T` is equivalent to `A.transpose()`.

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 `networkx` and print the corresponding adjacency matrix. Entering edges with a specified weight 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 = nx.Graph()`) or create a new one with a different name (e.g., `G2`) for this section.

You will notice that the adjacency representation in `networkx` does not double the weight of self-loops – the diagonal entries of your adjacency matrix are so far just 1. To see a bit more about how to access and modify the network, let's go ahead and explicitly make those self-loop weights equal to 2. We could do this in the initial `add_edge` function call, but let's observe another way to do this. Doing so reveals how `networkx` stores graph information. The network is stored in a three-layer dictionary, which is rather elegant and very "Pythonic"; it is also rather slow in terms of performance. This is why `networkx` is often not used for immense graph analysis. Recall, a dictionary is a collection of keys and values. Querying the key 2

```
G[2]
```

produces the corresponding value

```
{1: {}, 2: {}, 3: {'weight': 2}, 4: {}}
```

and reveals a new view of the network from the perspective of node 2. So the first layer of dictionary is nodes. `G[2]` is again a dictionary containing effectively the edges to neighboring nodes. In particular, it produces a dictionary of neighbor nodes, so the second layer is also nodes, but limited to nodes that have edges to the first node. The dictionary value above is, itself a dictionary and shows that node 2 is connected by an edge to nodes 1, 2, 3, and 4. In this case the key 1 corresponds to a value that is an empty dictionary (i.e., `G[2][1]={}`) because here I did not provide any additional information (e.g., weight) about the edge between nodes 1 and 2. In contrast the key 3 corresponds to a value that is a dictionary `{'weight': 2}` (i.e., `G[2][3]= {'weight': 2}`). This captures the fact that the weight of the edge between nodes 2 and 3 has a weight of 2. Thus, if we want to set the weight of the self-loop on node 2, we can use the command

```
G[2][2]['weight']=2
```

Use this to set the self-loops on nodes 2 and 6 before you print out your adjacency matrix.

Section 6.4: Directed Networks

Now 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 `nx.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 `networkx` 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 `networkx`; when the text uses A^T, use `A` in `networkx`. So, from now on a good practice is to always use the textbook convention, which means you should always use the transpose of the `networkx` adjacency matrix:

```
A = nx.adjacency_matrix(G).todense().T
```

In your printout, now using the textbook convention, the edge from node 6 to node 4 makes the entry in the fourth row and sixth column contain a 1. Notice that $A_{64}=0$ entry (sixth row, fourth column) 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.

When the original directed graph is unweighted, 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 = nx.read_gml('proofwikidefs_la.gml','name')
```

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 second parameter `'name'` indicates that the node names are specified by a parameter in the GML file called `name`, which overrides the default, which is `'label'`.

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.predecessors()` in the networkx DiGraph documentation. One subtlety to be aware of: 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
```

This function should use a graphical approach rather than the linear algebra definition (which we will used to validate your approach). 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 here it is important to remember 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)`

```
A = nx.adjacency_matrix(G).todense().T
C1 = numpy.dot(A,A.transpose())
Gc = cocitation(G)
C2 = nx.adjacency_matrix(Gc).todense().T
Cdiff = C1-C2
print('Difference between cocitation methods: %i' % Cdiff.sum().sum())
```

Because we're now using the NumPy module, you need to import the NumPy package. It is typically standard form to place all of your dependencies at the top of your file, so you can add the following line to the Networkx import statement at the beginning of your code.

```
import numpy
```

In the last line of the previous code block 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 – see the NumPy function `sum()`.

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 by printing both.

Now similarly implement a new function to produce the bibliographic coupling network from the original directed graph. In this case you may want to look at the `G.successors()` function of a `DiGraph` object.

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 `read_weighted_edgelist` command (note there is a separate command for reading edgelists without weights, `read_edgelist`):

```
G = nx.read_weighted_edgelist('acyclic1.edgelist',create_using=nx.DiGraph)
```

The last parameter of this function indicates that the edgelist should be imported as a directed network. 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 `networkx` 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. While `networkx` has a bipartite network type, we will not use it much because bipartite networks have some particularly rigid structures that make them less often relevant for large-scale analysis. Most of our use of bipartite networks in this course will be as theoretical tools. It is interesting to see how the concept of one-mode projection coincides with the graph transformations we discussed previously. We can import the actor-movie bipartite network instead as a directed graph in which edges point from actors to movies:

```
B = nx.read_gml('2013-actor-movie-bipartite.gml','name')
```

If we wanted to find the one-mode projection of this bipartite network onto the actors/actresses, would we use cocitation or bibliographic coupling? Write a sentence to justify your choice and then use this function to answer questions about the one-mode projection.

Consider the actors: Will Ferrell and Jason Statham. Print out their immediate neighbors in the one-mode projection, e.g.,

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

(this works because the node names 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.