Lab 2: Degree & Paths (Newman, 6.9-6.12)

I have provided some stub code in `lab2_stub.py`. At the top of the file add a commented out section (using #) and place your names and UTD netIDs there. Please make sure you label your print statements, so we know what you are printing. Some of the questions require a few sentences to answer it - for these you can answer either by typing your answer so it prints out in your program or you can attach a separate sheet of handwritten answers scanned as a PDF.

Sections 6.9: Degree

Degree is a measure of how connected a node is in the graph. Identify an example type of network of when it is a good thing to be a high degree node and an example of when it is a bad thing to be a high degree node in a network.

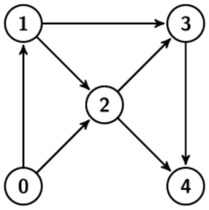
In one of the next labs, we'll be talking a lot about how we can identify important nodes in the network. One of the simplest ways, as we will see, is by using degree. However, there are some exceptions where this logic doesn't always work. Describe or draw an example of a network in which a particular node has very few connections, but it could be argued that it is a very important node. Justify your reasoning.

Explain what equation (6.25) in Newman actually means - i.e., what is a simpler way to explain what the two summations mean?

It is worth pointing out at this point that the `zen degree()` function returns just the number of edges a node has, not the sum of the weights as is described in the textbook.

Section 6.10-6.11: Paths & Components

You'll already be a bit familiar with paths. Sometimes we will call general traversals through the network as *walks*. Walks that don't intersect themselves (what the book calls *self-avoiding paths*) is what some people call paths.



Using the simple network diagrammed above, write down the adjacency matrix, `A`, (use the textbook convention, just so it matches what we'll do next). Then write down the adjacency matrix twice and multiply them to get A^2. Using equation (6.29) indicate what each nonzero element of A^2 means. Write down the path(s) that correspond to these nonzero elements: e.g., 1 → 3 → 4.

Now consider what happens to a vector `x` when it is multiplied by `A`. Set `x=[1,0,0,0,0]`^{T}, meaning that it is a column vector where the first entry is a one and the rest are zero. Multiply `Ax`, A^2`x=A(Ax)`, and A^3`x=A(A(Ax))`. What do these resulting column vectors represent (hint: look at the graph when you think about this)? Now do the same for `x=[0,0,1,0,0]`^{T}. Does your observation generalize?

Now let's jump into `zen` to work with paths some more. Along the way, I'll be introducing you to the visualizer that I made for `zen`, built on top of the d3js javascript library. Download the stub file and also the zip file of the visualizer. When you extract the visualizer it will have a folder called `zend3js`. My recommendation is to locate your Python work into one directory, say `NetworkModeling` (this name isn't important), with subfolders called `lab1`, `lab2`, etc (again these names aren't important) each of which holds your lab `.py` file and any data files you need for the lab. The visualizer directory `zend3js` (this name is important) should be inside your `NetworkModeling` folder as a sibling to the lab folders. Eventually you can also create a folder for your project that is also a sibling of the lab folders (because you may want to use the visualizer). If you want to put it somewhere else, that's fine, but you'll need to change the code a bit to find this `zend3js` folder (see the 3rd line of the stub code for this lab: `sys.path.append('../zend3js/')` indicates that this folder is just one folder up from where this lab `.py` file sits).

In the stub code I have already placed two functions for you to use. The first one, `square_grid(...)`, builds a network that is called a square lattice, which has a very recognizable structure. The second, `propagate(...)`, explores paths in a network by taking one "hop" at a time.

Below this, I have also already inserted some code. The line

```
d3 = d3js.D3jsRenderer(G, event_delay=0.03, interactive=False, autolaunch=False)
```

creates and loads the visualizer, which is defined in the file `../zend3js/d3js.py`. All that is required is to pass the network you want to visualize, in this case the empty network `G` we created in the line above. What is neat about this visualizer is that it listens to changes you make to the graph, so if you add/remove a node/edge, it happens in the visualizer. The `interactive` boolean flag indicates whether the visualizer should animate every change individually (`interactive=True`), or should wait for an explicit call to update, so changes can be made in bulk (`interactive=False`). The benefit for turning interactive off is that you can add/remove/edit a bunch of new nodes/edges without slowing the visualizer down. This visualizer can handle networks of up to around 500 nodes. There is no hard limit - things just slow down as you add elements to the visualization. With less than 500 nodes, animating adding all these nodes can still be a taxing job. When `interactive=False`, in order to see the changes you have made, you need to call `d3.update()`.

To learn more about the capabilities of the visualizer, you can take a look at the `demo.py` file in the `zend3js` folder. Two important options from the beginning are the `canvas_size` and `autolaunch` parameters. You can change the size of the "canvas" that is used in the visualizer so that it fits within your screen. By default `canvas_size=(800,600)`, which means the part that draws the network is restricted to 800 pixels by 600 pixels. This is a bit small for many of your laptop screens, so you can increase this size if you want it to have more room. By default the `autolaunch` parameter is set as `True`, which means that when you start the `D3jsRenderer` it will automatically open the (local) webpage we use to view the visualization. You can only have this open in one place (not on multiple tabs) otherwise the vizualizer gets confused. If it autolaunches each time you run your lab code, then you'll need to close the previous one each time. If you would prefer to open it yourself and leave it open, you can set `autolaunch=False` and then you need to double click `index.html` from inside the `zend3js` folder and leave it open. The visualizer should work in Chrome and Safari and probably other web browsers as well (unfortunately there are some issues with Internet Explorer, so Windows users probably want to set `autolaunch=False`). An example of setting these optional parameters is:

```
d3 = d3js.D3jsRenderer(G, canvas_size=(1000,800), autolaunch=False)
```

Grid Network

The next line makes a 12x12 lattice located nicely so you can see it easily. After this there is call to `d3.update()` so that the changes (adding all the nodes and edges) can be seen. With `index.html` manually open or with the `autolaunch=True`, run your python code and you should see your graph magically appear! If you're curious, you can change `interactive=True` to see what it looks like if you animate all the steps - it make get a little slow but it will get there eventually.

By hand, looking at the network, write down the degrees of the nodes (you can aggregate your answer, so write down how many nodes have degree `k`). Pick a way to visualize this aggregated degree data (a graph of some sort) - just draw something by hand.

Beneath I have some code (the next 4 lines) commented out that you can uncomment. The function `propagate` is defined above. Normally, I would have you actually write this function, but it gets a little more tedious to do than I would like in order to get the visualization to work just right. Look at the function implementation though for a moment. In line 3 of this function, we get the adjacency matrix of this network (recall that the adjacency in `zen` is the transpose to what is described in the book): `A = G.matrix().T`. The brains of this function are located on the 8th and 9th lines of the function where I have attached the comment `# the brains`. The second line does the matrix multiplication `Ax` and updates `x` with the result. The function `sign` just means that I keep the values of `x` as `0` or `1` (if any term is greater than zero, it is turned into a `1`). The loop means that we do this over and over again, so we are effectively computing A^kx, where `k=1,2,...,steps`.

So the call `propagate(G,d3,x,10,slp=1)` shows the result of the product $A^k x$ for `k=1,...,10`. The `slp=1` means that the program will take a 1 second pause between each step. To start the propagation from node index 0, we create `x` as a vector of zeros with **N=144** (the number of nodes) entries and set the 0th index to 1: `x[0]=1`. After this function call, we call the function `sleep` (imported from the module `time`), where the argument indicates how many seconds to pause before continuing - this makes the visualization easier to interpret if we put pauses between each section. Feel free to sprinkle these through your code to make the visualization happen at the right speed for your observations.

Run this code and watch the visualizer. Answer the following questions: Briefly describe why you see the pattern that you do and indicate why the process stops where it does. How could you make the propagation go further?

Directed Graph

Next, we'll continue to look at the directed network in the figure on the first page. You can now uncomment the next 4 lines of the stub file. The first thing to do is clear the visualizer so that we can move on to a new network. Then I have already saved the network above as an edgelist you can read in - notice in order to get a directed network from this edgelist, we need to set the `directed=True` flag in the import statement, otherwise it will be read as an undirected graph. Next, to inform the visualizer that the graph has changed (whether or not we use the same variable name), we pass the new graph using `d3.set_graph(G)`. Then we call update to see the graph appear. If you get tired of watching the previous steps, you can always comment them out, so the program goes directly to the part you're working on - just remember to uncomment these lines before you submit your code!

Now add code to propagate this network starting at node 0 for 10 steps. What is the end result? What is A^{10} (A is the adjacency matrix)? Describe why this is the case from the point of view of network paths and also from linear algebra (i.e., what special name do we give such matrices?).

Now let's look at components. In undirected graphs, components are quite simple - nodes that are connected to each other are in the same component. In directed graphs, because of the direction associated with the edges, we can talk about components (weak connectivity that simply ignores the direction of the edges) or in- and out-components (connectivity that depends on the directions of the edges). Let's look at out-components. One way to out-components is to propagate the network forward from a node until the set of nodes reached doesn't change anymore - this will be the out-component of the graph. Before we do this look at the network and determine the out-component of node 1 (note that if you hover your mouse over the node in the visualizer, it will indicate the node name - the object value). Now propagate the network starting at node 1. For this you will want to set the optional flag `keep_highlights=True` in the `propagate` function call. This keeps all the nodes that have ever been highlighted. If we didn't use this, we would get a result similar to what happened in the paragraph above (which wouldn't be very useful!). Run this code - is the out-component what you predicted?

Without changing the `propagate` function, how could we use it to find in-components instead of out-components?

Strongly connected components combine the ideas of in- and out-components. What is the size of the largest strongly connected component?

E. coli Protein Network

Now we are going to load in the directed network that represents the connectivity of protein pathways in the E. coli bacteria. The description of this network is:

```
Transcriptional regulation networks in cells orchestrate gene expression. In this network the nodes are operons, and each
edge is directed from an operon that encodes a transcription factor to an operon that it directly regulates (an operon is
one or more genes transcribed on the same mRNA). The transcriptional database contains 577 interactions between 116 TFs
and 419 operons. It was based on an existing database: (RegulonDB). We enhanced RegulonDB by an extensive literature
search, adding 35 new TFs, including alternative sigma factors, and over a hundred new interactions from the literature.
The dataset consists of established interactions in which a TF directly binds a regulatory site.

Citation: S Shen-Orr, R Milo, S Mangan, U Alon, Network motifs in the transcriptional regulation network of Escherichia
coli. Nature Genetics, 31:64-68 (2002).
```

This network has some nice structure for us to investigate. Uncomment the first block of code in this section and run it to see the network. Some other fun features of the visualizer are that you can: zoom (scroll), pan (click and drag the white background), move nodes (click and drag a node to pin it at a specific location), unpin nodes (double click on the node to unpin).

Start by finding the out-components of nodes with index 2 and 16. Have your program print out the size of the component. Because this network is larger, updating the highlighting at every step gets slow, so it would be better to calculate all the steps and just highlight the end result. You can do this by setting the flag `update_at_end=True` as a parameter to the `propagate` function. The number of steps is unknown because we don't know how large the out-components are. What is the minimum value of `steps` that guarantees that you will find the entire out-component?

The function `propagate` returns the final `x`. In order to figure out the size of the out-component, you need to determine how many entries in `x` are nonzero. Numpy has a handy function for this called `where`. If you have a array `x`, the statement `where(x>0)` will return the indices where `x` is greater than 0. It returns it in a tuple, so you need to use `where(x>0)[0]` to get the array of indices (which is the first element of the tuple). In between these two cases, you will need to clear the highlighting that the `propagate` creates. You can use the function `d3.clear_highlights()` and then `d3.update()` to do this.

Our last item is to find and display the diameter of this network. The diameter is the longest shortest path. This is a confusing definition at first. It means we calculate, for every pair of start node and target node, the shortest path between these nodes. Then from all of these shortest paths, we pick the one that is the longest. This gives us a sense for the size of the network in terms of how many hops it takes to get around the network.

I already have some code to get you started, which you can now uncomment. The function `diameter` computes the diameter of the network (big surprise!). This gets us the length of the longest shortest path, but doesn't help us visualize it on the network. To do this we call the "all-pairs" version of Dijkstra's algorithm. Look at the documentation. `D` is a matrix that has the length of the shortest path between any two nodes such that D_{ij} is the shortest distance from node `i` to node `j`. The matrix `P` holds each step of the path from every node to every other node. We first find which node pairs are included in shortest paths that are equal to the diameter: `uidx,vidx=where(D==d)`. Since there are multiple shortest paths with length equal to the diameter, we will just pick the first one: `uidx[0]` and `vidx[0]`. We use the handy `zen` function to convert the matrix `P` into a path. So `path` is an array that has the node indices from the start node to the target node and each hop in between. Since the diameter is `d`, we expect this array to have `d+1` nodes in it. I leave it to you to loop through these nodes and find the edge indices (look at `edge_idx_` in `zen.DiGraph`) that correspond to this path and to highlight them using `d3.highlight_edges_(...)`. Note that this function takes a list of edge indices to highlight.

Section 6.12: Flows & Cut Sets

Flows and cut sets can be used to identify bottlenecks in the network and can be used a one way of establishing the robustness of the connectivity of a graph. The nodes or edges in the cut set can be interpreted as important because they are a vulnerable point in the network - if they get removed, the function of the underlying system can be deeply affected!

For a network of fixed number of nodes, design (weakly) connected, directed, unweighted graphs that achieve the best case and worst case running time for the minimum cut (augmenting path algorithm). Describe these graphs in words. Define functions to make these networks (something like `best_graph(n)` and `worst_graph(n)` which take in the number of nodes in the graph). Choose `n` to be somewhat large (1,000) and observe the computation time required on these two types of graphs. The function you need is in the `zen.algorithms.flow` module. You can measure the time it takes to compute something (in seconds) by using the [`time` module](#) in Python:

```
import time
start_time = time.time()
call_function()
print time.time() - start_time
```

Now let's take a look at a project management technique that relies on some of the algorithms that we've discussed. Program Evaluation and Review Technique (PERT) is a way of viewing the order and dependencies of the steps of a project as a network. You may be familiar with an Gantt chart, which identifies the expected start and end dates of a particular task in a larger project. Undoubtably, sometimes these dates get delayed because a particular piece of the project takes longer than expected. PERT analysis is a bit more robust to this because all durations are relative to the previous predecessor tasks. PERT takes the Gantt chart type data and assigns durations to each activity and the activities that must be completed before another activity can begin. For example, the activities of reading the assigned reading and listening to the lecture should really precede your attempt to do this lab. Each of those have an expected duration and this ordering can be captured by placing a directed edge from the reading activity to the lab activity as well as from the lecture activity to the lab activity.

One of the useful tools that comes out of PERT analysis is the fact that we can compute the longest path in the network from the beginning node to the ending node and this represents the expected time to completion. This path (or possibly paths if there are multiple) is called a *critical path* because the overall project timeline will be delayed if any of these subtasks get delayed. Colloquially we might call these the "rate limiting steps" of the project. In a large project eyeballing this is difficult, and this is where network analysis helps us out!

First, reason why this graph of activities should be an acyclic directed graph. Explain why cycles in a project activity network would be a bad way to organize it. This is important because in order to find the *longest* path in the network, the graph must not have cycles. For general graphs, the problem of finding the longest path is intractable (in that there exists no polynomial-time algorithm, i.e., $O(n^k)$, to solve this problem). We'll talk about why this is in more detail later (it is related to Hamiltonian paths), but if we know that the graph has no cycles, we can analyze the graph `-G`, which is the original graph `G` with all the edges negated (e.g., an edge with weight 6 becomes an edge with weight -6). Dijkstra's algorithm no longer works on graphs with negative weights but another shortest path algorithm, called Bellman-Ford, is designed to handle such graphs provided that there are no cycles with negative weight (a cycle of negative weight is one in which the sum of the edges in the cycle is negative). Let's look at the following example project related to laying down a pipeline at a construction site.

Activity	Description	Duration	Prerequisites
1	Lead time	10	-
2	Move to site	20	1
3	Obtain pipes	30	1
4	Obtain valves	20	1
5	Lay out pipeline	7	2
6	Dig trench	25	5
7	Prepare valve chambers	18	3,6
8	Cut specials	9	3
9	Lay pipes	20	3,6
10	Fit valves	10	4,7,8
11	Concrete anchors	12	9
12	Finish valve chambers	8	10,11
13	Test pipeline	6	10,11
14	Backfill	10	11
15	Clean up	3	13,14
16	Leave site		12,15

I have loaded this data into a graph file called `pert.gml`. When we construct the activity network from this data table, all edges leaving node 1 to other nodes have a weight of the duration of activity 1 since it takes 10 units of time to complete activity 1 and move on to the next activities (in this case 2, 3, and 4). Using the Bellman-Ford shortest path algorithm, find the length and activity sequence of the critical path in this activity network.

While cut sets are not conventionally used in PERT analysis, calculate the minimum cut set and the cut set weight from "Lead time” to "Leave site”. Notice that the minimum cut algorithm is only defined for positive edge weights. By default the minimum cut function uses `capacity=’unit’` (so assuming an unweighted network), but you can change this to `capacity=’weight’` instead. Can you think of a potential use-case for analyzing the minimum cut of an activity network?