Computer Science 112 - Fundamentals of Programming II
Lab Project 11
Due on Github 11:59pm Monday 8 April

For this lab we will explore the small-world network issues that I presented informally in class. Specifically, we will use our **UG** class from the previous lab to replicate a classic "six degrees of separation" result from some code I found online.

Part I: Small-world networks with networkx

Googling **small-world networks in python** produced a nice AI-generated result; namely, a Python script showing how to use the networkx Python library to generate a small-world network in a single line of code, and another small script showing how to plot the network using the matplotlib library. So the first thing you should do for this lab is to install these two libraries on your computer (or, if you're using the lab computers, let me know if they're not installed). Once you've installed them, combine the two scripts into a single script **smallworld1.py**. Running this program should yield the relevant statistics (mainly, the average shortest path) and should also plot the graph for you.

Once you've got your smallworld1.py script running, comment-out the code that draws the graph and reports the clustering coefficients, leaving the line that prints out the average shortest path length – *i.e.*, the "degrees of separation" result that interests us. Can you make some predictions about what will happen for certain values of *K* (number of local neighbors) and *P* (probability of replacing a local neighbor with an arbitrarily distant one)? For example, when I set $K = 2$ (each node has a neighbor on either side) and $P = 0$ (there are no random long-distance neighbors), I got an average shortest distance of slightly over 25. This makes sense, because your farthest neighbor is 50 steps away, your closest neighbor is 1 away, and if you only have one neighbor on either side, you have to take 50 steps to get to your farthest neighbor. Think of a few "extreme" *P* and *K* values like this, and report the average shortest distance value for a network built with each pair of values. That will complete your smallworld1.py network for this part.

Part II: Building our own small-world networks with our UG class

To start, copy your **ug.py** file to wherever you're working on this current lab. Then create a **smallworld2.py s**cript that starts with `from ug import UG`.

Looking over the main() code in my ug.py, I see that my UG constructor requires me to input a potentially large set of vertices followed by a large set of edges. This is okay to write by hand for up to 10 or so vertices, but for our hundred-vertex small-world application it's going to be too cumbersome. So your first task in **smallworld2.py** is going to be figuring out how to create the list of vertices and edges to pass to the UG constructor. Since a network with N=100 is far too big to print out using a connectivity-matrix format, I suggest starting smaller, say N=10, and passing an empty list as your edges. For consistency with Part 1, I found it simplest to make my vertices by a list of numbers, which I converted to strings so they'd work with our existing UG: `['1','2','3','4','5','6','7','8', '9', '10']`. Later you can figure out how to generate such a list for a larger *N* like 100.

Now it's time to figure out how to specify our list of edges to pass to the UG constructor, so that it they will be similar to what we get with the **watts_strogatz_graph()** function in the networkx library. Googling **watts_strogatz_graph** gave me a nice explanation of how to lay out the topology (edges) of the network. As usual, rather than trying to do the whole thing at once, I broke it down into simpler parts,each of which I checked by printing out the UG at each stage:

1. Make a ring by connecting each node to its two closest neighbors (one neighbor on each side). Since we don't care about weights, I just used a weight of 1 for all edges.
2. Instead of just two neighbors, make the ring work for *K* neighbors.
3. With probability *P*, replace each existing connection with a connection to an arbitrary other node.

Once you've debugged your code for a network with size *N*=10, run your Floyd algorithm on it to see the shortest paths.

Now it's time to scale it up to the size of the original network *N*=100. Obviously you shouldn't try to print out something so large, but you should still be able to run the Floyd algorithm to get the matrix of shortest distances (I had to modify my **ug.py** a bit to get the matrix instead of its string representation). To get the mean shortest distance (the "degrees of separation"), you can loop over the rows and columns of the matrix, accumulating a sum of the matrix entries, and then divide by the appropriate amount at the end. Once I had that working for the original settings *K* = 4, P = 0.1, I set K = 1 and P = 0 to see whether I could get the exact same result as in my smallworld1.py script. Finally, I repeated the same final tests in smallworld2.py; i.e., try out and report the results for a few different *K, P* pairs.

What to turn in to Github

Turn in your **smallworld1.py**, **smallworld2.py,** and updated **ug.py**.