Overview In this homework, I'm asking you to implement a class for a certain mathematical structure called a graph. There are many methods that I want you to implement. I hope that the math is self-contained, but if anything is unclear or if you are interested to learn more, please let me know and I will work to clarify. I have also included extra exercises to include a few other aspects of the course we have covered.

Notice that I have not included any starter code or testing code. Please test/debug your code as you write it.

1 Graphs

A *graph* is a mathematical structure that describes a set of objects, called *vertices* or *nodes*, and relations between pairs of the objects, called *edges*. We denote a graph G with an ordered pair of sets (V, E) that describe the nodes and edges of G. Figure 1 shows a graph on six nodes. Graphs are very useful objects that represent many real world data sets, including social networks (people are nodes and friendships are edges) and transportation networks (cities/points of interest are nodes and roads are edges).

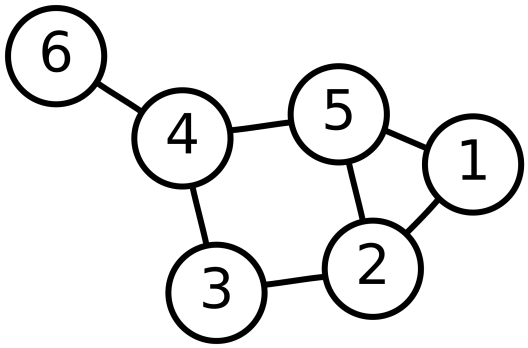

Figure 1: From Wikipedia page for graph (discrete mathematics)

We may represent a graph with a matrix called the *adjacency matrix*, A_G. For a graph $G = (V, E)$ with $|V| = n$, then $A_G \in \mathbb{R}^{n \times n}$ has the following structure:

$$A_G(i, j) = \begin{cases} 1, & \text{if } (i, j) \in E \\ 0, & \text{otherwise} \end{cases}$$

We will be working exclusively with *undirected graphs*. That is, edges are bi-directional; if $(i, j) \in E$ then $(j, i) \in E$. Furthermore, we will assume that only a single edge may exist between any two nodes and no self-edges exist (edges of the form (i, i)). These graphs are called *simple*. Notice that these assumptions mean that A_G will be symmetric and $A_G(i, i) = 0$ for all $i \in V$. For example the following is the adjacency matrix for the graph depicted in Figure 1.

$$A_G = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

In this question, you will create a class in Python for graphs. Throughout this question, you may assume that the user input into all methods is valid.

1. Create an initialization method for your class that takes as input the adjacency matrix for the class. Keep the adjacency matrix of the graph as an attribute. You may add additional attributes as you see fit. (10 points)

2. Create a method that adds an edge to the graph. It should take two nodes as input and create an edge between those two nodes. (5 points)

3. Create a method that adds a node to your graph. The method should take as input an optional list of nodes with which the new node shares an edge that should default to nothing if no list is provided. (5 points)

4. Create a method that returns the adjacency matrix for the graph. (5 points)

5. The *Laplacian matrix* of a graph is another useful matrix that represents a graph. One way of writing the Laplacian is the following: $L_G = D_G - A_G$, where A_G is the adjacency matrix of G and D_G is the degree matrix of G. D_G is a diagonal matrix such that $D_G(i, i)$ is the degree of node i (the number of edges it is adjacent to) and $D_G(i, j) = 0$ if $i \neq j$. Create a method that forms and returns the Laplacian matrix of G. (10 points)

6. The set of *neighbors* of a node $i \in V$ for a graph $G = (V, E)$, is the set of nodes that share an edge with i (i.e. neighbors of $i = \{j : (i, j) \in E\}$). We may further define the *neighborhood* of a set of vertices $S \subseteq V$ as the union of all neighbors of vertices in S. Create a method that given a set of vertices $S \in V$ computes and returns the neighborhood of S in G. (10 points)

7. One way to think of addition on graphs is the union. That is, we may think of the addition operation as given $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$, $G = G_1 + G_2 = G_1 \cup G_2 = (V, E_1 \cup E_2)$. Note that here we are assuming that the vertex set of the two graphs that we wish to add are the same and the sum graph has an edge (i, j) if either of the graphs we are adding does. Overload the addition operator for the class (`__add__` method) to reflect this definition of addition. (10 points)

8. We often care about the eigenvalues of a matrix or graph. An eigenvalue, eigenvector pair (λ, v) of a (square) matrix A is a scalar λ and vector v such that $Av = \lambda v$. The set of eigenvalues/eigenvectors describe the action and conditioning of a matrix. The power iteration (https://en.wikipedia.org/wiki/Power_iteration) is a simple algorithm that finds the dominant eigenvalue of a matrix and it's associated eigenvector (or a vector in the associated eigenspace). Create a method that takes as input a square matrix and a tolerance and uses the power iteration to find the dominant eigenvalue and associated vector. This method does not have to be within your graph class, but we will use the computation for other methods of the class.

   Note that you should be using numpy arrays, but do not use numpy functions that compute eigenvalues for you (such as eig). The tolerance is a stopping criterion. That is, if the tolerance is ϵ, then the computation should terminate if $\|x_{k+1} - x_k\|_2 < \epsilon$ where the vectors are at iterations k and $k + 1$. Hint: if you did exercise 9.4, you should be able to reuse code. (15 points)

9. Create two functions that utilizes your power iteration method that return the dominant eigenvalue for each of the Laplacian and adjacency matrices for the graph. (5 points)

10. An Erdos-Renyi random graph is a random graph parametrized by two values: n and p. A random (n, p) Erdos-Renyi graph is a graph on n nodes such that each possible (of the $\binom{n}{2}$) edge exists (independently) with probability p. Create an Erdos-Renyi class which inherits from your graph class. The class should have a constructor that takes two inputs, n and p, and creates a random (n, p) Erdos-Renyi graph. All other methods for the class may be the same as the parent graph class. (10 points)

2 Shorter questions

1. Now that you have created your graph classes, we are going to use them to run a simulation. The second smallest eigenvalue of the Laplacian of a graph G ($\lambda_2(G)$) is often called the *algebraic connectivity* of

the graph. That is, it provides some measure of how well-connected G is (note: a graph is said to be *connected* if all nodes are reachable from all nodes). If G is not connected, $\lambda_2(G) = 0$ (note: $\lambda_1(G) = 0$ for every graph - if you care to see this, you can easily observe that the all ones vector is in the null space of G) and larger values of $\lambda_2(G)$ intuitively correspond to graphs with many different pathways between all pairs of vertices.

Create a script (you can and maybe should create functions to help you do this) that for a fixed n, creates many (choose a good number) Erdos-Renyi graphs for p values distributed between 0 and 1 (for example, $p = 0, 0.05, 0.1, \ldots, 0.95, 1$). Compute the mean of $\lambda_2(G(n, p))$ for each value of p you have used (that is, if you have 1000 graphs with for $p = 0.05$, you should compute $\lambda_2(G)$ for all of them and then compute the average of those). Note that you may use built-in numpy functions to compute eigenvalues for this question. Finally, use matplotlib to plot the average of λ_2 as a function of p. (10 points)

2. Exercise 12.8 (Fibonacci sequence) (5 points)