CS5016: Computational Methods and Applications
Assignment 2: Networks, Random Graphs and Percolation

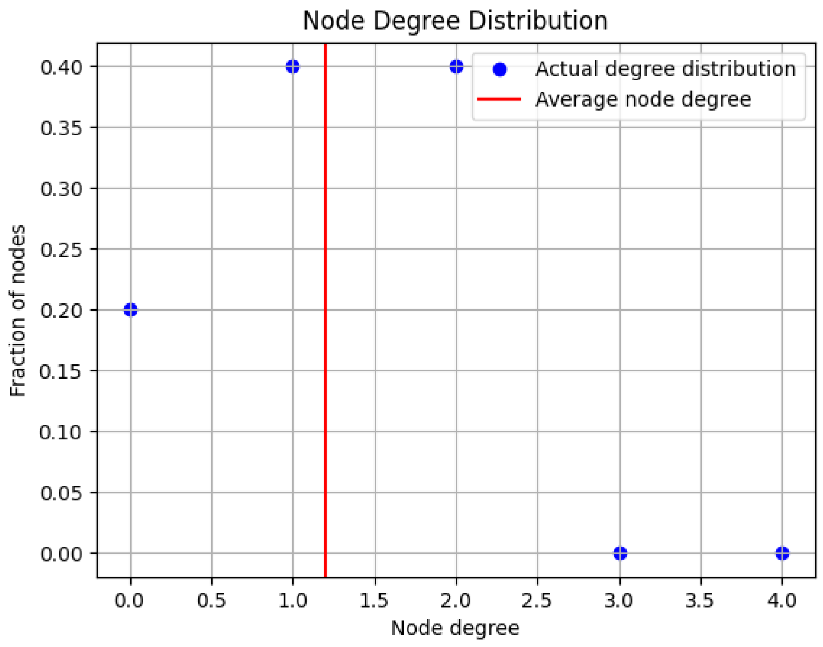
Sujit Mandava
112001043

Q1. The class UndirectedGraph is an implementation of an undirected graph using adjacency lists. The __init__ method takes an optional argument n, which is the maximum number of nodes. When given, it checks the validity of the input and raises an appropriate exception whenever needed. For every node, it creates an adjacency list to represent the edges coming out of it, which is stored using sets. If n is not given, it creates a free graph. This graph initially has no nodes or edges, they can be added up to any index. When n is given, a graph with nodes up to index n and no edges is initialized. For such graphs, nodes past the index n cannot be added.

The method addNode allows us to add nodes to the graph. If the graph is free, nodes can be added to any index. However, for graphs with a maximum number of nodes, nodes with a higher index than the maximum number of nodes cannot be added.

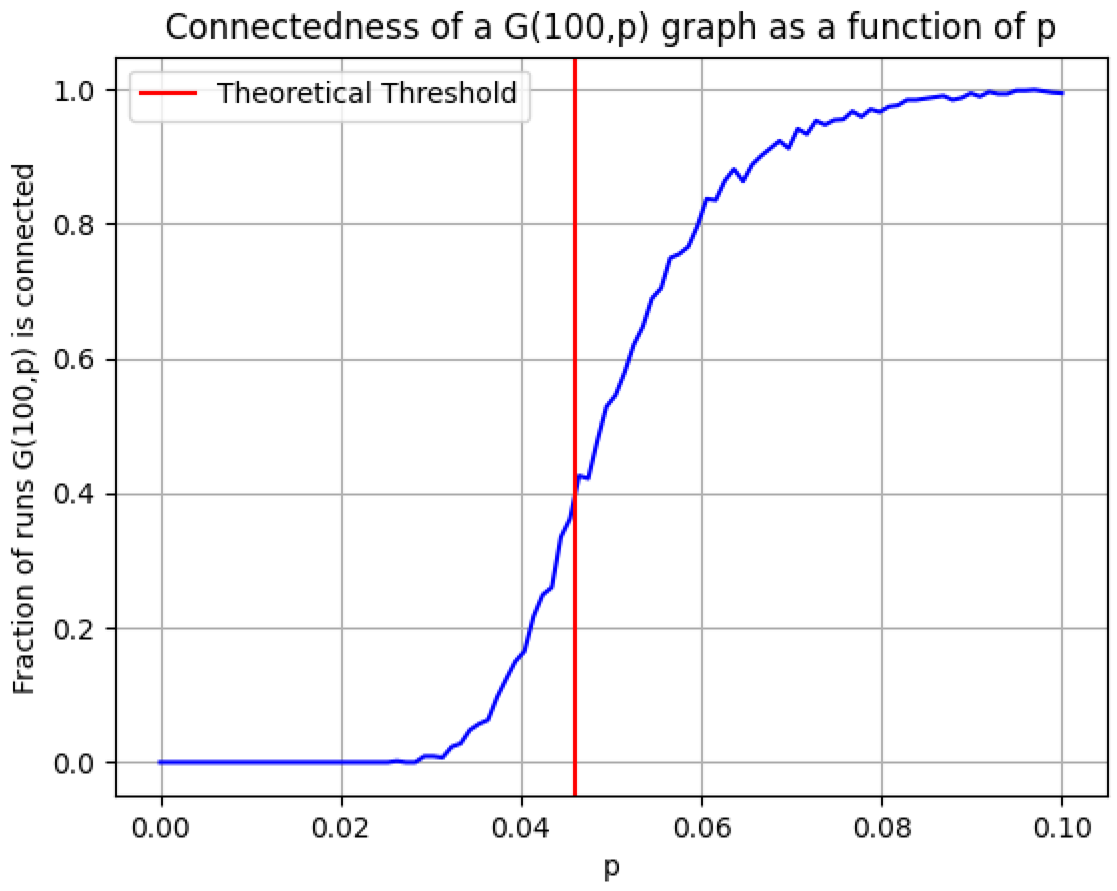
The method addEgde adds edges to the graph. If the nodes are not present, it calls addNode to add these nodes and then puts each other into their adjacency lists.

The method plotDegDist plots the degree distribution of the given graph. Sample output may look like this:



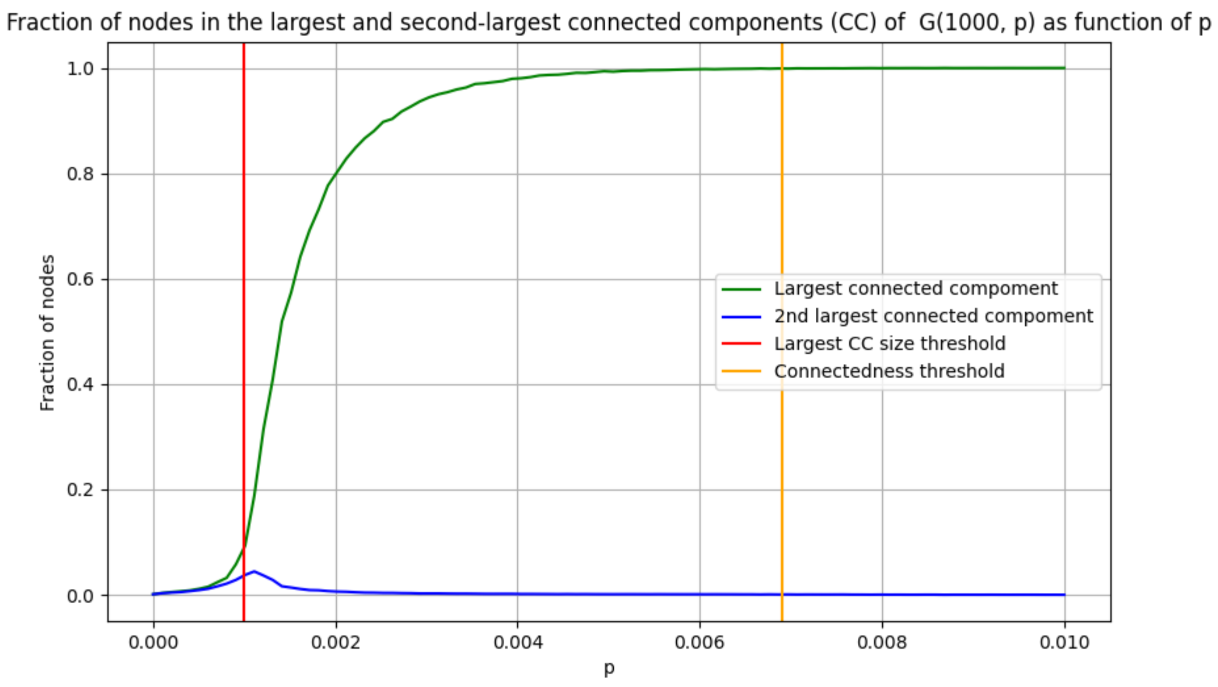
Q2. The derived class ERRandomGraph has an extra method: sample(). This method takes an input, p ranging from 0 to 1. It then uses p as the probability with which an edge might exist in a graph and generates random edges in it.

Q3. The method isConnected() is a simple implementation of breadth-first search. The idea is that if the graph is connected, then irrespective of the starting node we choose, we will get only one single component. If not, then the graph is disconnected.
We take any node and start BFS from it. At the end of the search, we check the number of vertices the algorithm has visited. If it visits all, then the graph is connected.
The method verifyConnectedStatement() verifies the following statement given in the problem statement using the above method. It takes an array of values of p between 0 and 0.01. It then makes an ERRandomGraph for each p 1000 times, and plots the average number of times the graph G(1000, p) is connected. The output graph looks like the following:

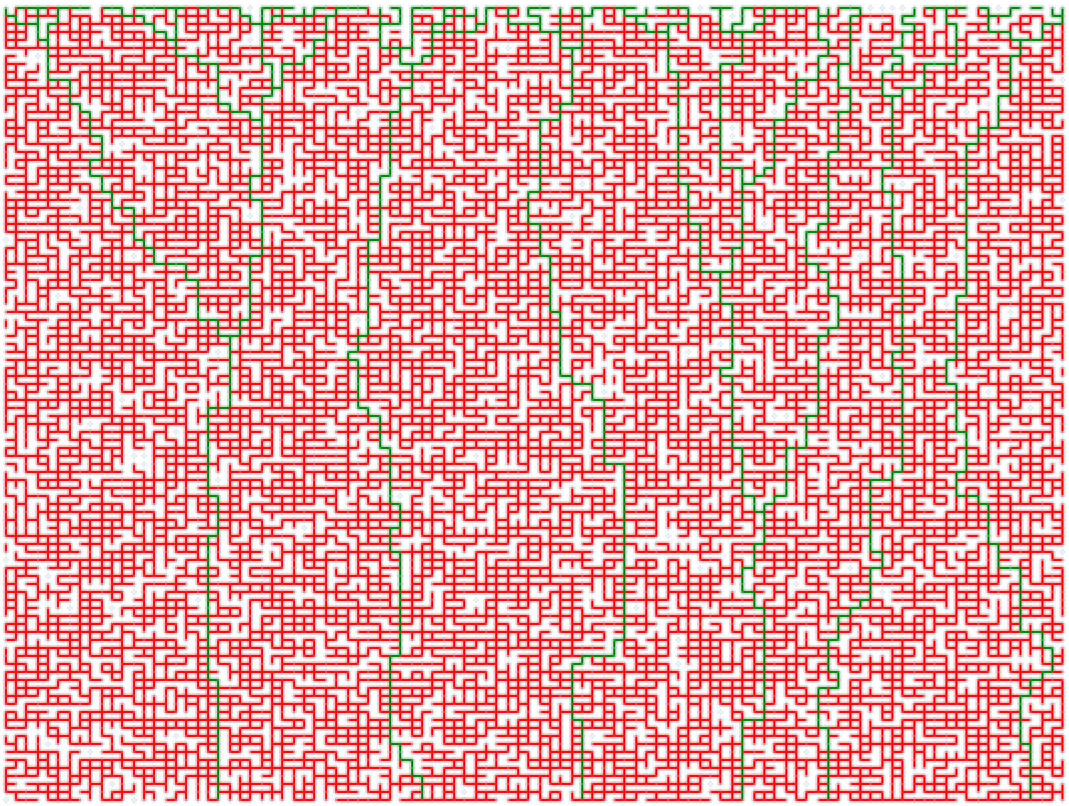


Q4. The method oneTwoLargestComponents() is used to find the largest two components in the graph. It is also a BFS algorithm. This time, however, we run the search for every node. Before running the search, we check if the current start node has been visited by a previous iteration of the search. If it has been visited, then the component has been found and the search can move to

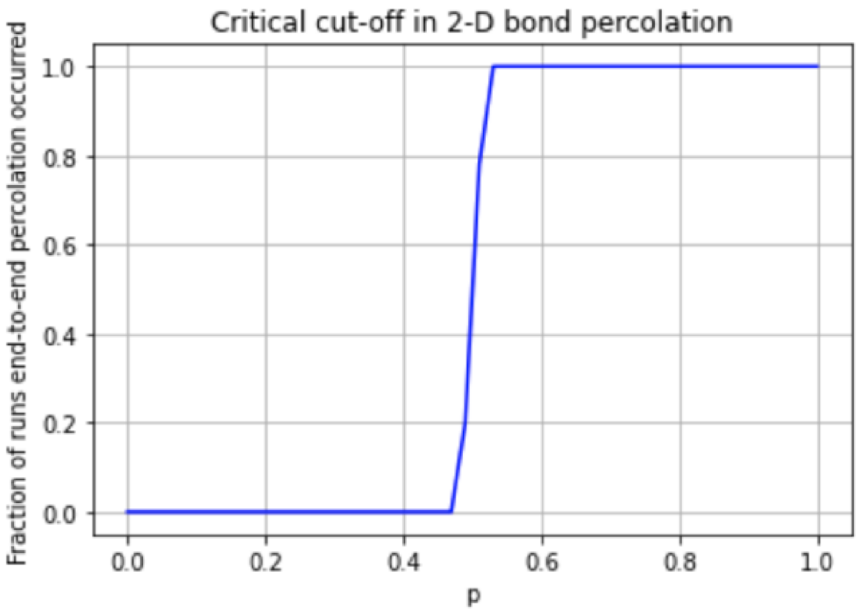
the next algorithm. We collect the sizes of all such components(one per search iteration) and then pick the largest two components.
The method verifyComponent() verifies the statement in the problem statement. It first takes an array of p's between 0 and 0.01 to generate ER Random Graphs. Then, for every p, it creates a graph and finds the fraction of nodes in the largest and second-largest components for 50 runs and averages it. It then collects and plots these values. The graph looks similar to:



Q5. The class Lattice was created using tools from the NetworkX library. I used the grid_2d_graph method to create an empty grid. To simulate percolation, we take an argument p which is used to sample the edges for every node. Then, we use BFS starting from every node at the topmost level. We create edges from these nodes with probability p to nodes that or on the same or at a lower level than them. For every node, we keep track of the parent node and the depth at which it is present from the start. This way, we can easily tell whether there is a path to the bottom, and which paths connect the top to the bottom. The method existsTopToDown uses the former, while the method showPaths uses the latter.
existsTopToDown simply checks whether there is a node whose depth is equal to the maximum possible depth, i.e, n in an nxn lattice. This can be done by making the BFS algorithm return the maximum depth of a visited node.
showPaths marks all the paths present in the lattice by backtracking from the last node in the search to the initial node via the list of parent nodes being maintained. In case the path connects top to bottom, it marks them in green, otherwise red. The lattice would look similar to:



Q6. The method verifyPath is used to verify the statement in the problem statement. It takes an array of p's to simulate percolation in the lattice. Then, for every p, it simulates the percolation 50 times and counts the number of times it reaches the bottom. Then it collects the average number of times there is a path to the bottom and plots the graph. The graph looks similar to:


Critical cut-off in 2-D bond percolation
Fraction of runs end-to-end percolation occurred
p