CS5016: Computational Methods and Applications

<u>Lab Report - 2</u>

111901030
Mayank Singla

First of all, we need to handle exceptions in all the questions. So to avoid repetition of code, I made a Decorator Factory function for class methods called `handleError`, which simply wraps the class method in a `try-except` block and handles and prints the error as in the expected format. This decorator forwards all the arguments it receives to the actual method and also returns the value returned by the method so that the functionality of the method doesn't break. I am using the same decorator factory function in all of my questions for all the class methods.

Q1.

I am creating an `UndirectedGraph` class to represent an undirected graph. I am creating a constructor which takes the number of nodes in the graph as an optional parameter. If it is not provided, the graph is free. I am representing the internal graph data structure as an adjacency list. I am also storing the number of nodes, edges, and the maximum number of nodes in the graph.

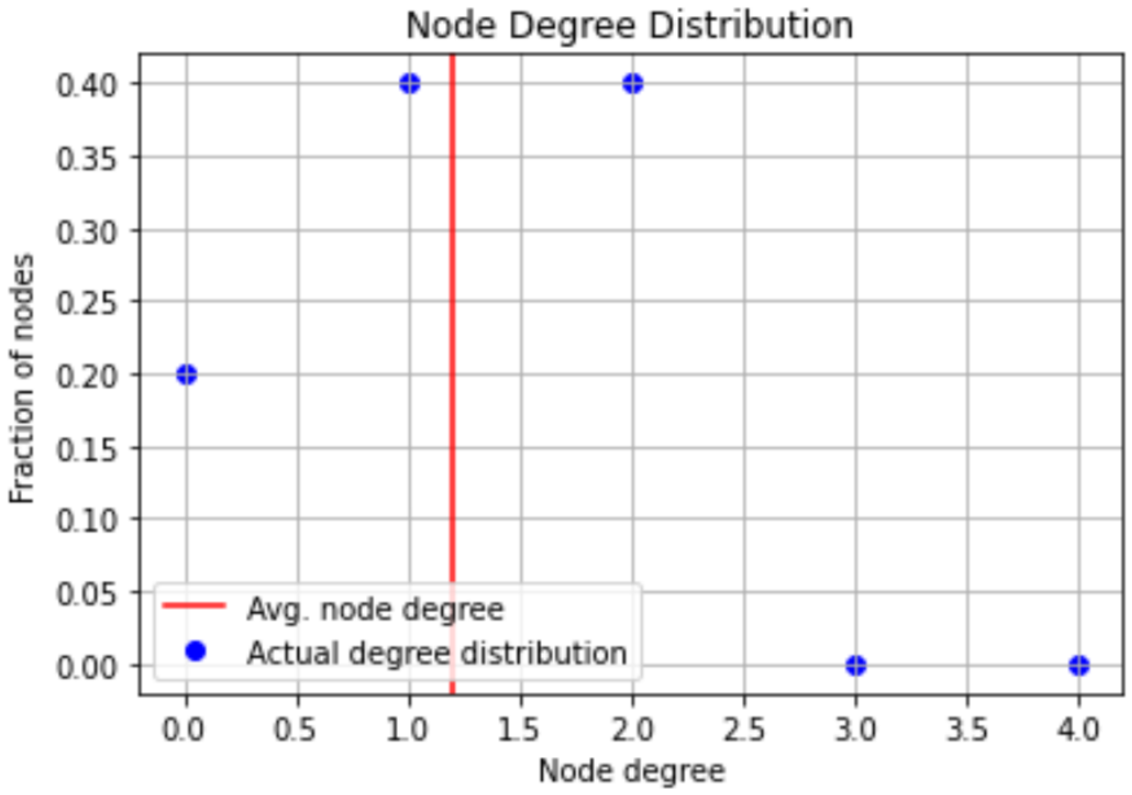
Next, I am creating an `addNode` method to add a node to a graph, which can only be possible in the case of a free graph. For adding a node, I need to validate the correctness of the value passed a parameter, and for doing that I am creating a `_validateNode` method to check for validity and raise exceptions if an invalid node input value is received. Similarly, for adding an edge, I am creating a method `addEdge` to add an undirected edge to the graph and it also corresponding nodes to the graph if they are not already present, and again to validate the correctness of an edge, I am creating a `_validateEdge` method to raise exceptions for invalid edge input.

To overload the plus operator, I am using the dunder method for class `__add__`, and based on the input I receive, node or edge, I am adding it to the graph. It calls the `addNode` and `addEdge` class methods already implemented to add them to the graph.

For the graph to be printable, I am using the dunder method for class `__str__` and returning the string in the expected format to print it.

Next, I am creating the method `plotDegDist` to plot the degree distribution of the graph. For this, as a helper method, I am creating a `_calcDegreeDist` method which returns a dictionary that has the key as its degree and value as the number of nodes in the graph with that degree. I am plotting the degree distribution curve for each degree from `0` to `n - 1` where `n` is the number of nodes in the graph.

Here is an example of the plot I plotted for the sample test case given the question.

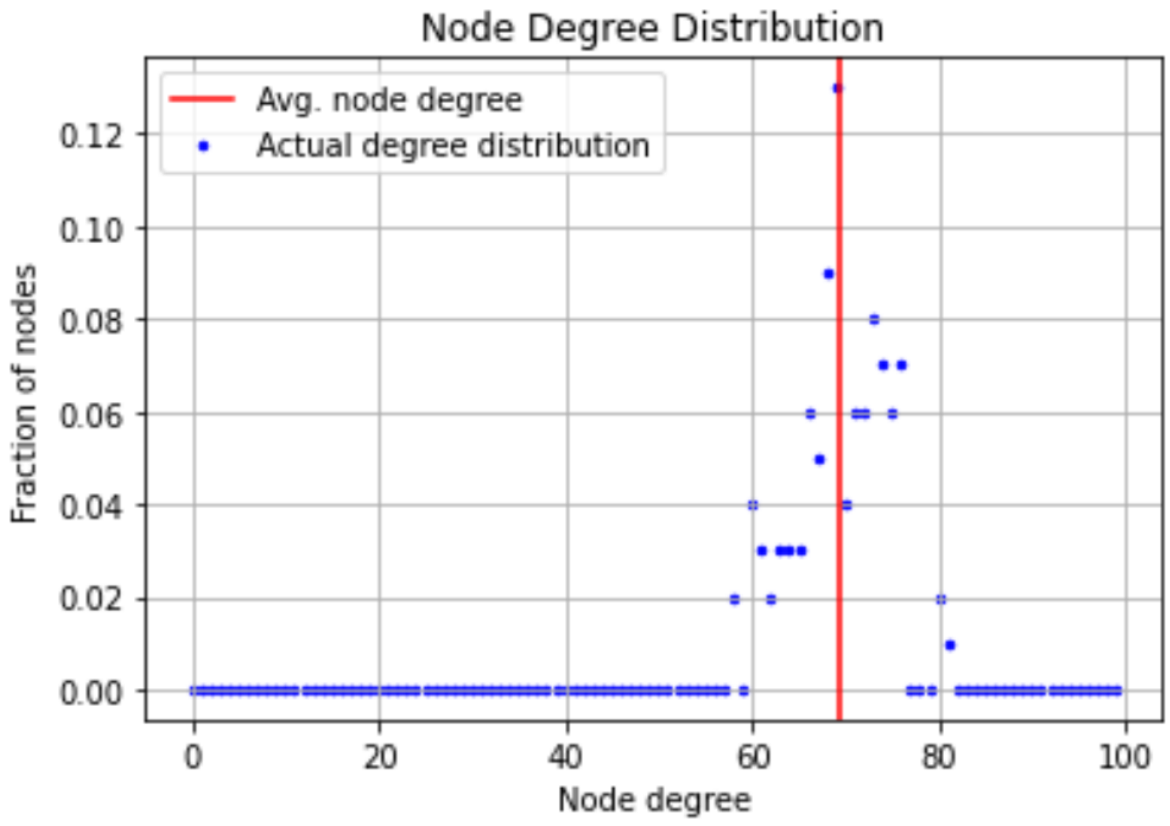


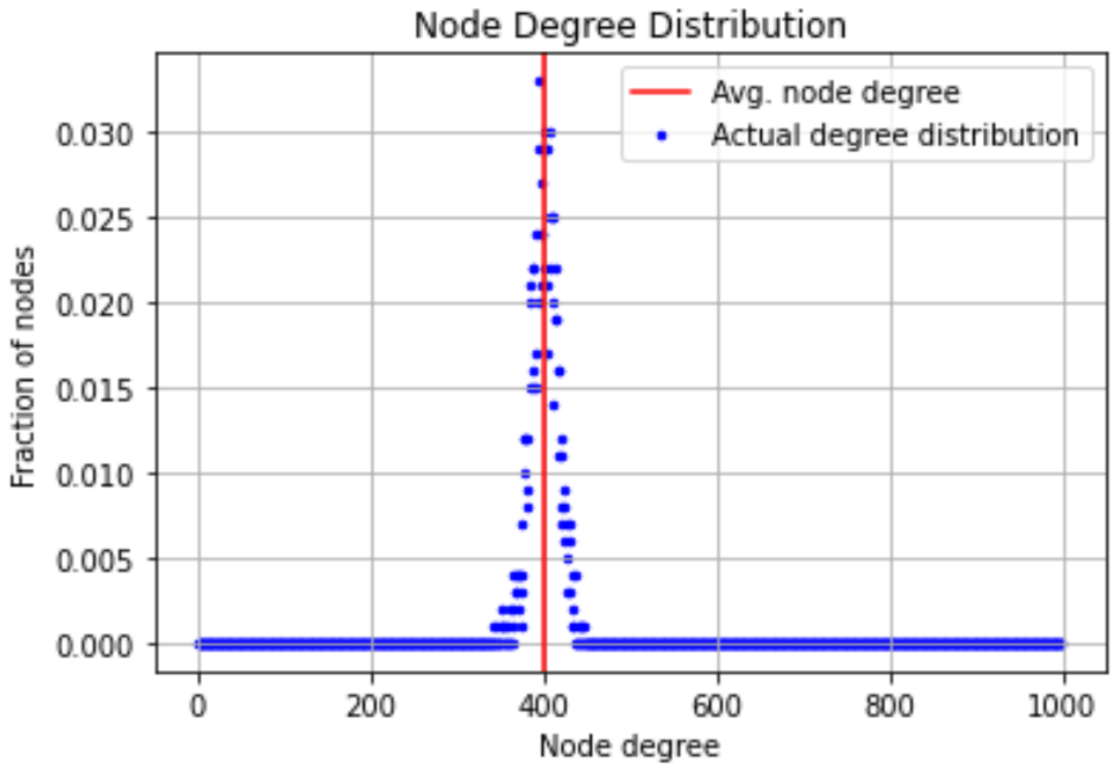
Q2.

I am creating an `ERRandomGraph` class that inherits the `UndirectedGraph` class. It has a constructor method that takes a mandatory argument which is the number of nodes in the graph and it simply calls the `super` constructor of the parent class and passes it the number of nodes as the argument.

Next, I am implementing a method `sample` that generates a random graph `G(n, p)` for the input probability `p`. It loops through all the edges of the graph possible and generates a random number for each edge from `[0, 1)` and if that number is less than `p`, then we add the edge b/w those two nodes. This is how I am ensuring that the probability of occurrence of an edge is `p`. For generating the random number, I am using python's in-built `random.random()` function.

Here are the degree distributions of the random graphs I created on the given sample test cases.



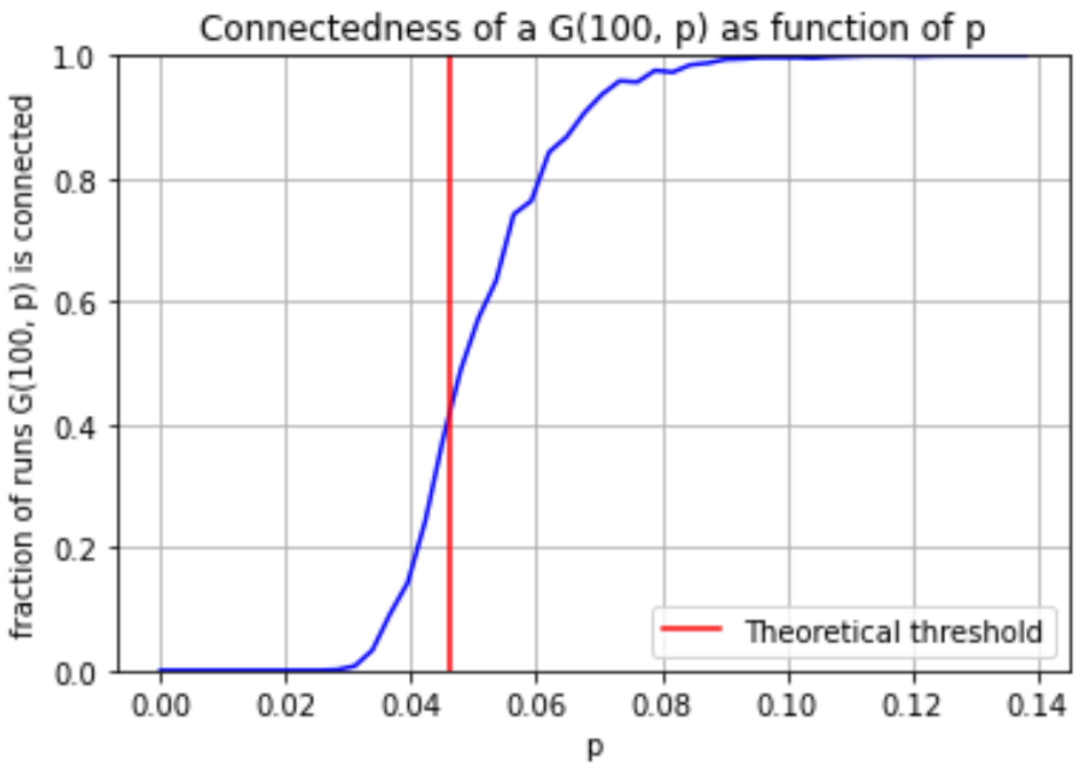


Q3.

First, I am implementing the method `isConnected` for the `UndirectedGraph` class which checks if the graph is connected or not. As a helper method, I am creating an `_isConnectedHelper` method which simply does the traditional BFS from a starting vertex of the graph and marks all the vertices in its component as visited. For doing BFS I am using pythons' built-in `queue.Queue` data structure. The logic for connectedness is that I should run BFS only once from a starting vertex, and only in that execution, all the nodes should be marked as `visited`. If I need to execute BFS multiple times that means, the graph is not connected.

Next, I am creating a `verifyERConnectednessStatement` method that verifies and visualizes the **Erdos-Renyi** model that the graph `G(n, p)` is almost surely connected only if `p > log(n)/n`. I am creating it as a general method that will work for any value of `n`. I am creating some number of probability points to plot on the x-axis between some minimum and maximum probability and for each probability point, I am sampling a random `G(n, p)` graph for some number of runs. For each run, I check for the connectedness of the sampled random graph, and each probability, I maintain the count of the number of times the sampled random graph was found to be connected to get the

final fraction of runs for which `G(n, p)` was connected. Then I am plotting all the required things on the plot.
Here is plot I got for `n = 100`, `number of runs = 1000`, `number of probability points on x-axis = 50` in around **3 minutes** of execution.



Q4.

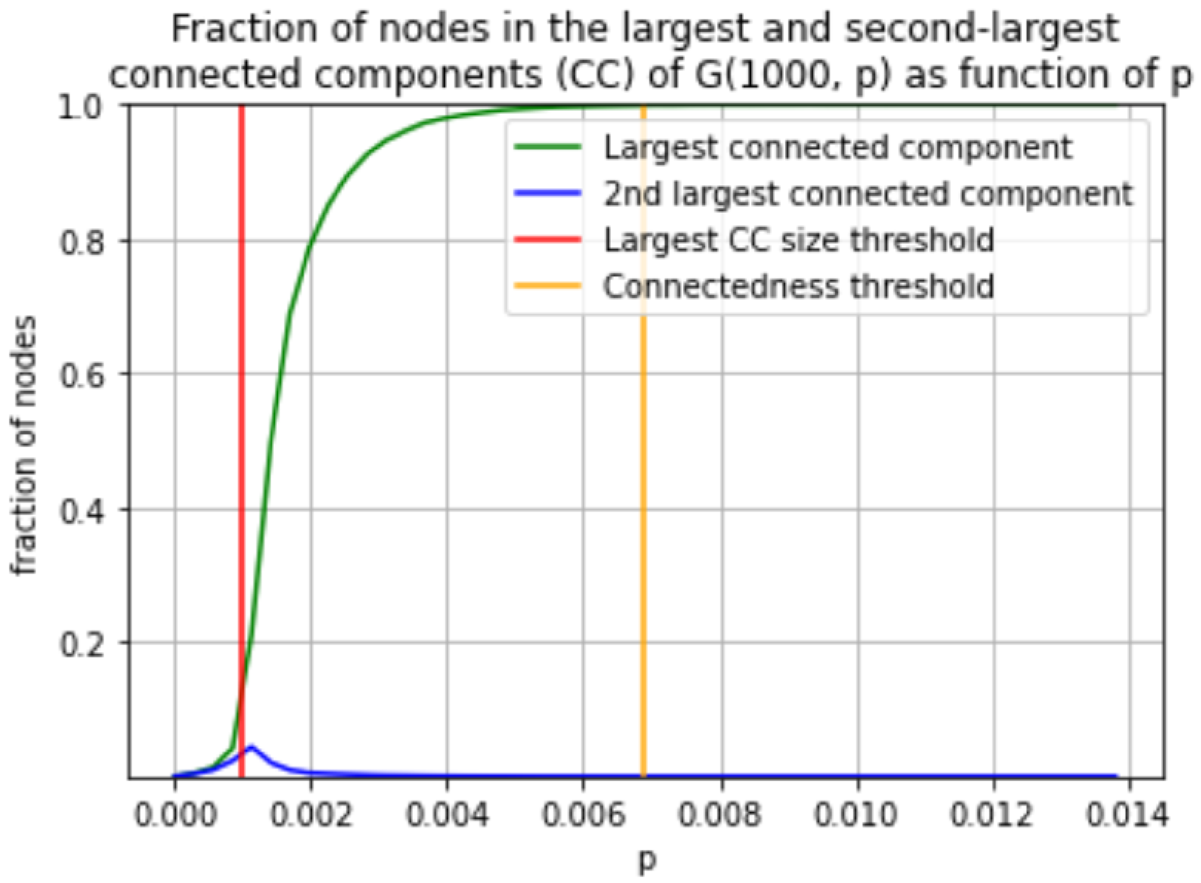
First, I am implementing the method `oneTwoComponentSizes` for the `UndirectedGraph` class which returns the size of the largest and the second-largest connected component in the graph. As a helper method for this method, I am implementing an `_oneTwoComponentSizesHelper` method which simply executes BFS from a starting vertex and returns the size of the component that starting vertex belongs to. I am doing this for all the components and keeping track of the values of the largest and the second-largest components and finally returning it.
Next, I am implementing the method `verifyERGiantComponentStatement` which verifies and visualizes the Erdos-Renyi Giant Component statement which states that:

- If `p < (1 / n)`, the Erdos-Renyi random graph `G(n, p)` will almost surely have only small connected components.

- On the other hand, if `p > (1 / n)`, almost surely, there will be a single giant component containing a positive fraction of the vertices.

For this, I am creating a certain number of probability points to be plotted on the x-axis and for each probability, I am sampling a random `G(n, p)` graph for some number of times and evaluating the fraction of nodes in the largest and the second-largest connected component of it. Finally, I am plotting all the required things shown in the question. The method I am creating will work for any general value of `n`.

Here is plot I got for `n = 1000`, `number of runs = 50`, `number of probability points on x-axis = 50` in around **3 minutes** of execution.



Q5.

I am creating the `Lattice` graph to represent a lattice graph. I am using python's [NetworkX]() module to build a square lattice graph. I am storing the position of nodes of the graph based on matrix indexing notation. Also, I am making the adjacency list of nodes for the graph which will be initially empty and will get renewed whenever we call the `percolate` method of the graph.

Next, I am first creating three helper methods. The first is `_drawNodes` which takes the position of nodes, their size, and width as input and draws them on the lattice graph using the built-in function in the NetworkX module which is `draw_networkx_nodes`. The second is `_drawEdges` which takes the position of nodes, the edges, their color, and width(optional) as input parameters and draws them on the lattice graph using the built-in functions in the NetworkX module which is `draw_networkx_edges`. The third helper method is `_getLatticeEdgeList` which converts the adjacency list of edges stored into the required format of edge list expected by the NetworkX module to plot them.

Next, I am implementing the `show` method to display the lattice graph which calls the helper methods `_drawNodes` and `_drawEdges` to display the lattice graph.

Next, I am implementing the `percolate` method to percolate the lattice graph with input probability using bond percolation. I am first clearing the adjacency list of vertices for the lattice graph. Then I have created three nested utility functions. The first one is `_isProbable` which returns `True` if the random number generated using `random.random()` is less than the input probability. The other two functions are `try_add_edge_right` and `try_add_edge_down` which tries adding an edge between the corresponding nodes into the adjacency list with the input probability. Finally, I am traversing over all the possible edges and calling those utility functions to try adding an edge between them with the input probability.

Next, I am implementing a helper method `_getPathEdges` that is given input the current node, the root node, and parent dictionary which keeps track of the parent of

each node and it backtracks from the current node to the root node and returns the list of edges in this path.
Next, I am also implementing another helper method `_bfs` that is given a starting node and an optional parameter `onlyCheck`. If that optional parameter is true, this method only returns whether there exists a path from the starting node to a node in the bottom-most layer of the lattice graph. If that optional parameter is false, this method either returns the list of edges in the path from the starting node to the farthest node if the lattice graph is not percolating or it returns the list of edges in the shortest path from the starting node to a node in the bottom-most layer of the lattice graph if the lattice graph is percolating. I am doing all this using the traditional BFS execution using a queue data structure and maintaining the required quantities.

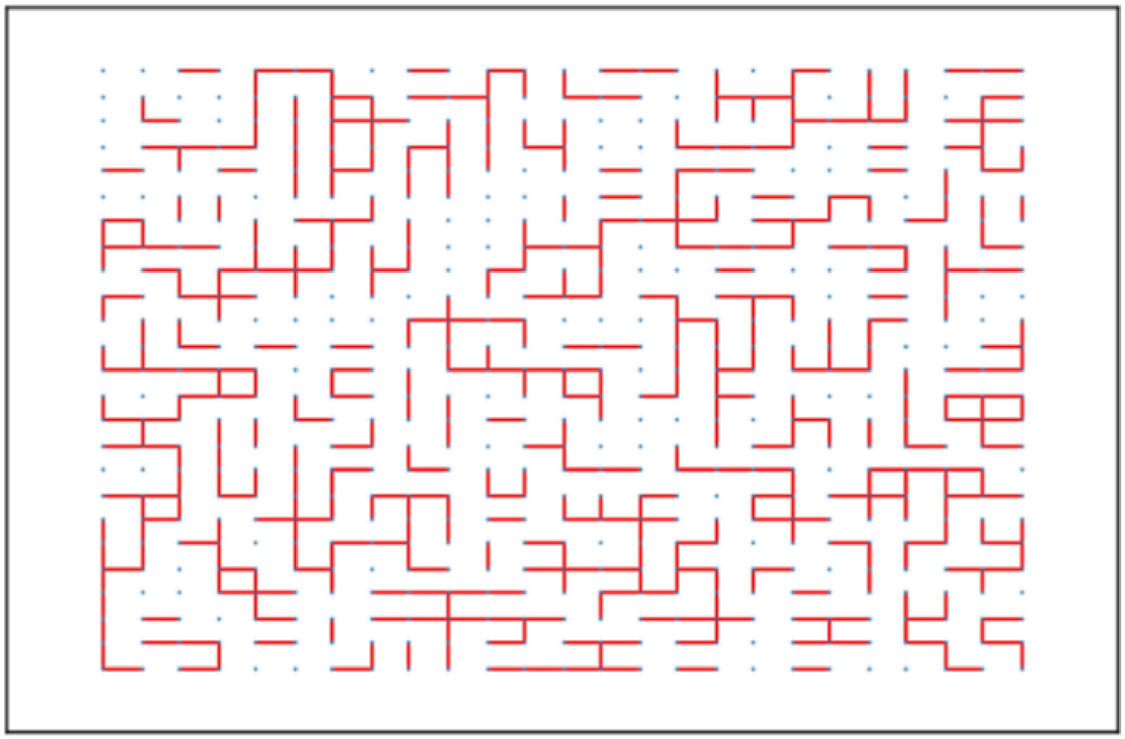
Then, I am implementing the `existsTopDownPath` method which returns True if the lattice graph is percolating i.e. there exists a path from a node in the top-most layer to a node in the bottom-most layer in the lattice graph. I am simply calling the `_bfs` method from each node in the top-most layer with that optional parameter being True to find if I can find such a path from any node in the top-most layer.

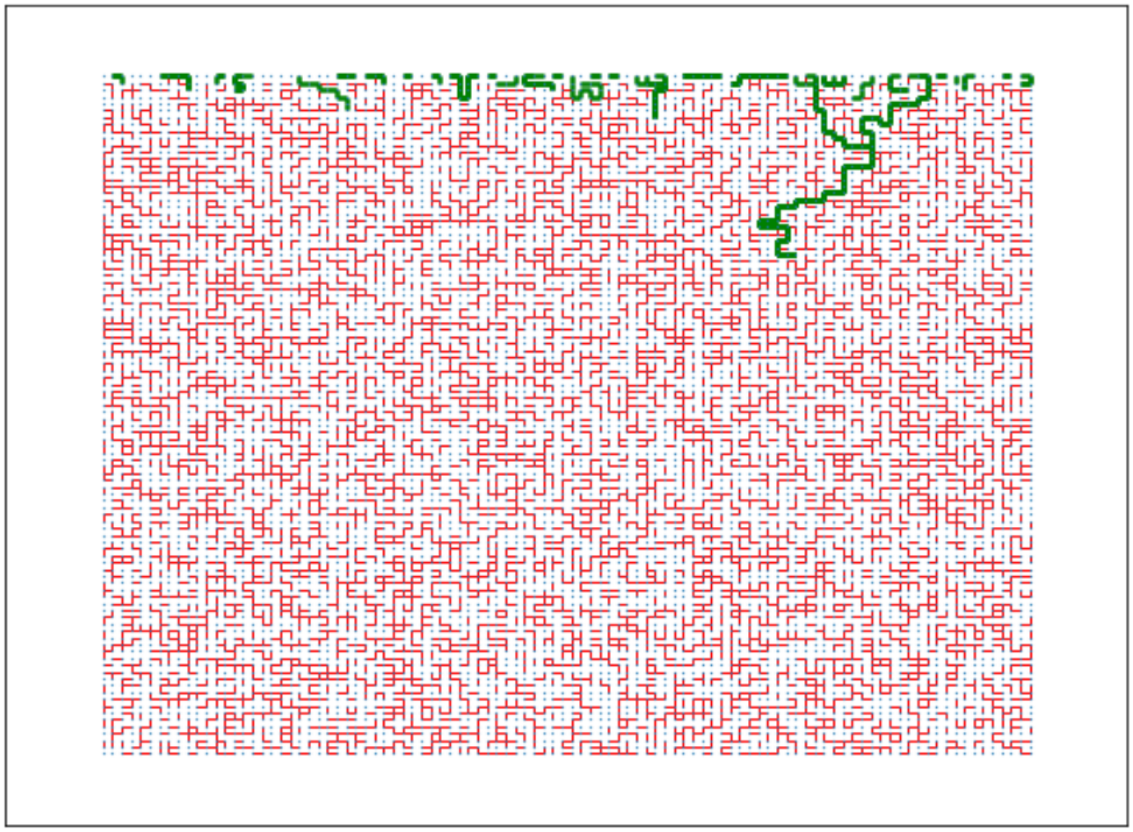
Finally, I am implementing the `showPaths` method which for every node `u` in the top-most layer, does either of the following.

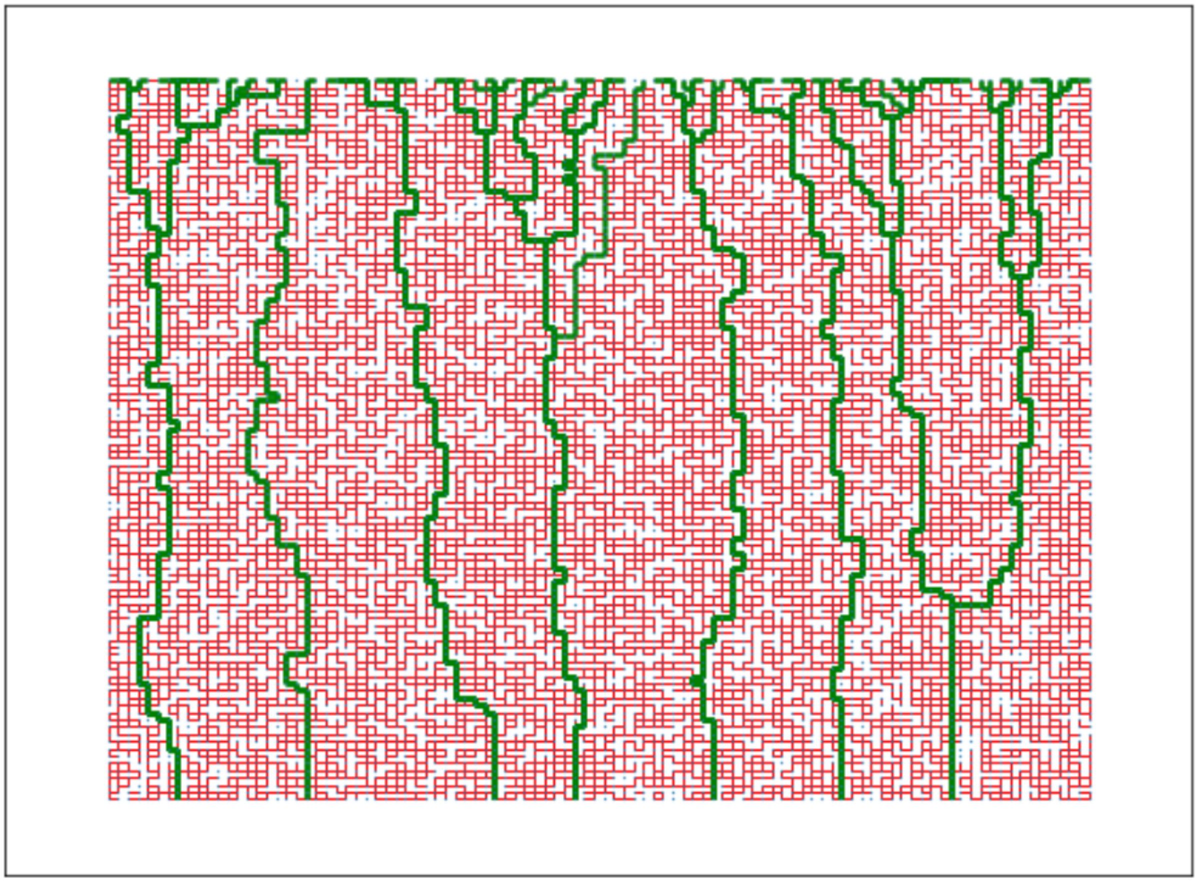
- If there is no path from `u` to nodes in the bottom-most layer, display the largest shortest path that originates at `u`.
- Otherwise, display the shortest path from `u` to the bottom-most layer.

For this, I am simply calling the `_bfs` method from every node `u` with the optional parameter being false to get the list of edges in the path as mentioned above and drawing all those edges. I am also calling the `showPath` method before this to display the nodes and default edges of the lattice graph.

Here are some of the outputs I got for the sample test cases mentioned in the question.





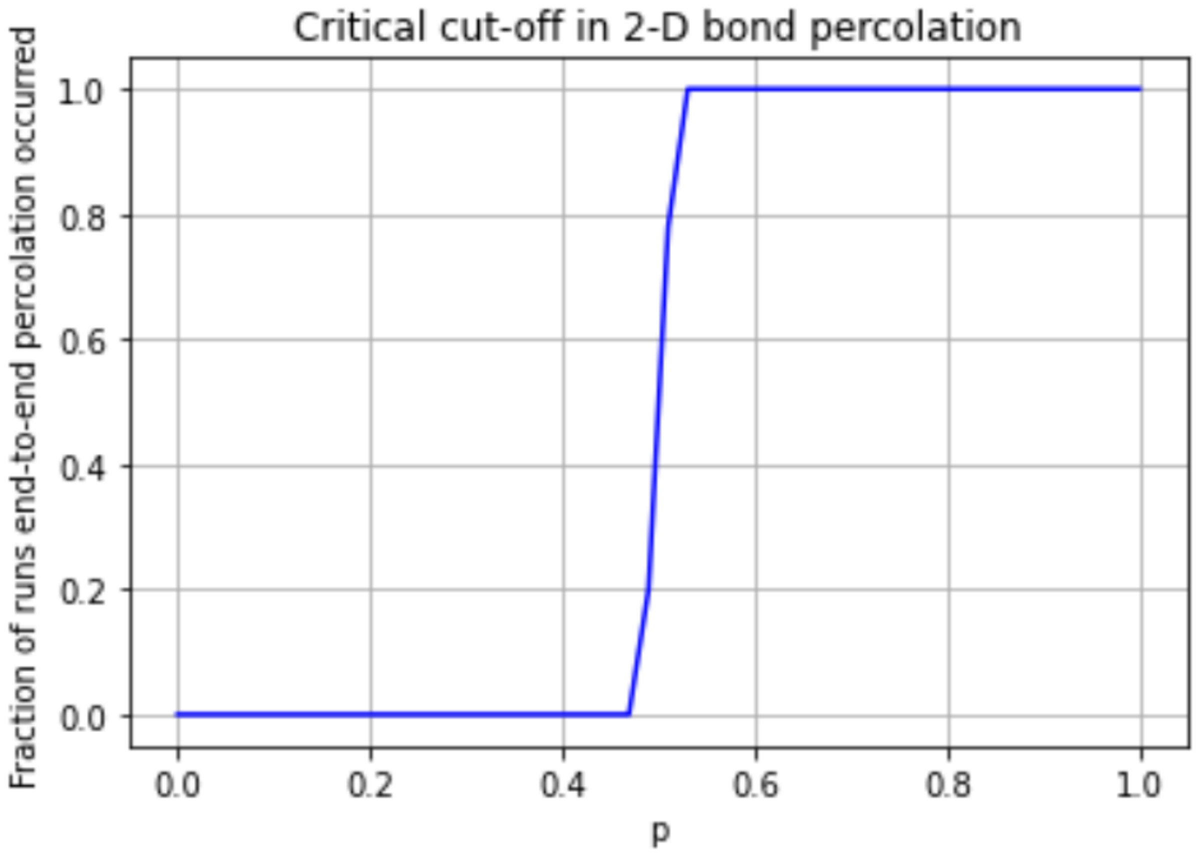


Q6.

I am implementing a method verifyBondPercolationStatement which verifies the following: *"A path exists (almost surely) from the top-most layer to the bottom-most layer of an nxn grid graph only if the bond percolation probability exceeds 0.5"*

For this, I am creating a certain number of probability points between `[0, 1]`, and for each probability point, I am percolating (calling the `percolate` method) the lattice graph and calculating the fraction for which end-to-end percolation occurred by maintaining the count for the same. And, then finally plotting the required curve.

Here is plot I got for `n = 100`, `number of runs = 50`, `number of probability points on x-axis = 50` in around **2 minutes** of execution.


Critical cut-off in 2-D bond percolation
Fraction of runs end-to-end percolation occurred
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
p