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

112001051
VM Sreeram

Question 1

It is assumed that only simple graphs are allowed — no self-edges or multi-edges. This was assumed because node degree (from plot) is upto total number of nodes-1, which is invalid for non-simple graphs.

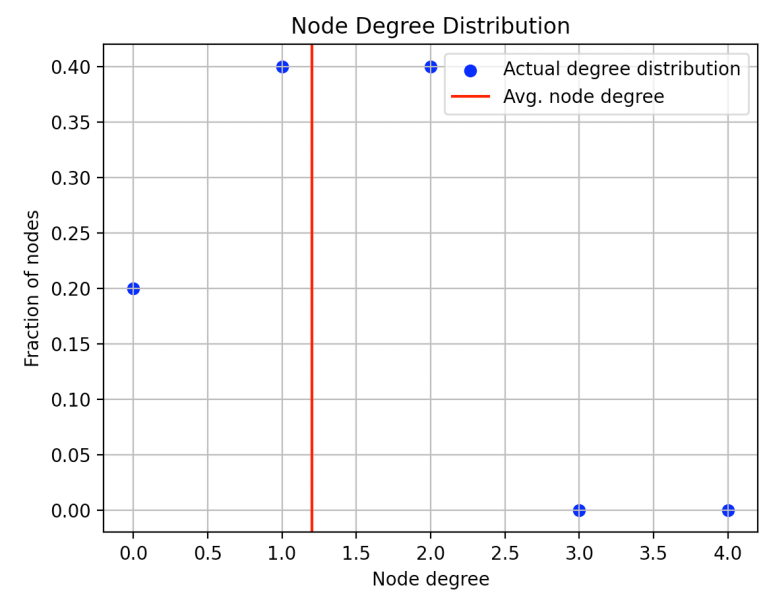
The adjacency list is implemented as a Python dictionary. Key is added whenever a node is added to `UndirectedGraph`, with value being an empty list, denoting the neighbouring nodes to the node corresponding to the key. Only valid indexed nodes are allowed to be added. Operator + is overloaded using `__add__` method. Object is made printable by defining `__str__` method in the format given in the problem statement. Node degree distribution is computed for node degrees from 0 to maximum possible degree corresponding to number of nodes in the graph.

Example input

```
g = UndirectedGraph()
g = g + 100
g = g + (1, 2)
g = g + (1, 100)
g = g + (100, 3)
g = g + 20
print(g)
g.plotDegDist()
```

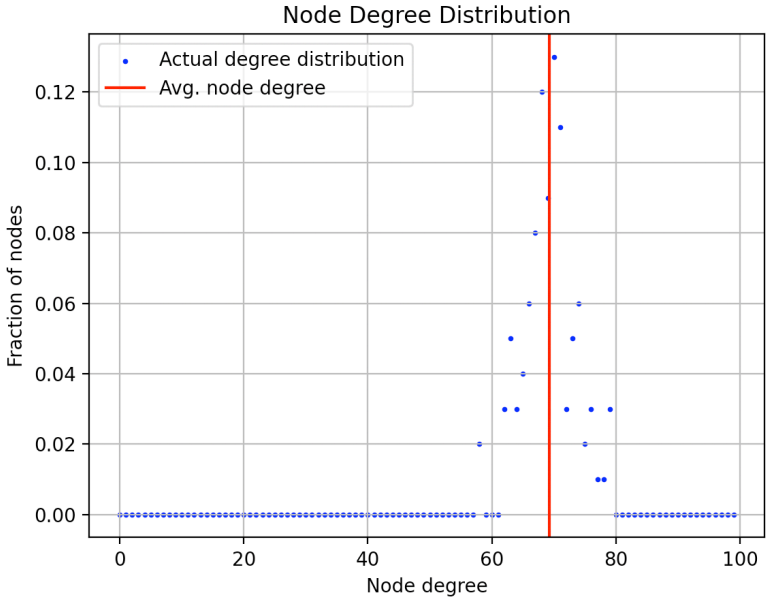
Output

```
Graph with 5 nodes and 3 edges. Neighbours of the nodes are belows:
Node 100: {1, 3}
Node 1: {2, 100}
Node 2: {1}
Node 3: {100}
Node 20: {}
```



Question 2

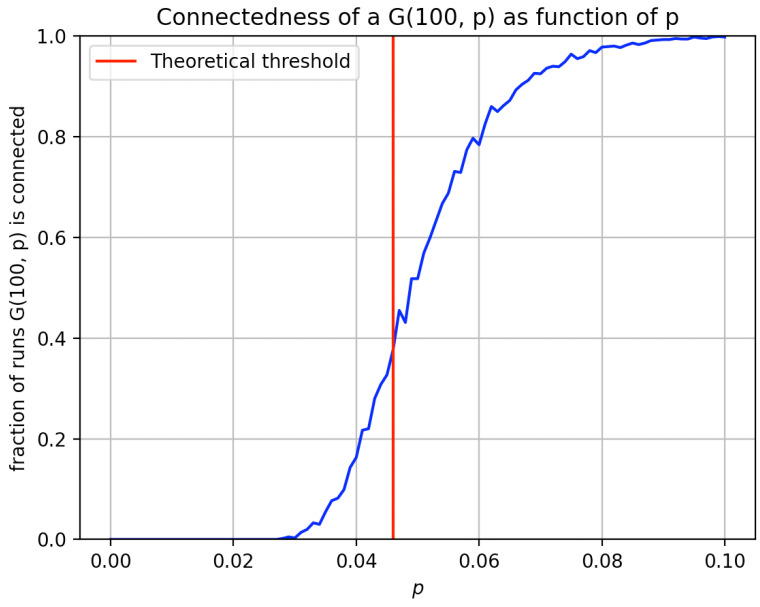
The class `ERRandomGraph` inherits the class `UndirectedGraph`. One positive integer argument is compulsory to the constructor, meaning the graph should have fixed number of vertices. `random.random()` is used to generate random number for sampling.



Node degree distribution of Erdős-Rényi random graph with n=100, sampled at p=0.7

Question 3

A method `isConnected` is added to the class `UndirectedGraph`. Let us follow definition of connection to be having a path connecting any two vertices in the graph. This definition holds vacuously for null graphs, and hence my code says null graph to be connected. If the graph is not a null graph, standard Breadth First Search (BFS) is executed to check for connectivity.



Output for verification part

The function `verify_er_connectivity()` is implemented to verify the statement given in the second part of the problem that Erdős-Rényi random graph G(100, *p*) is almost surely connected only if p > ln(100)/100. The function in my code is specific to n=100, because setting complicated limits upto where to compute *p* is not elegant.

To plot the graph, the fraction of runs G(100,*p*) is connected is computed for values of *p* from 0 to 0.1 in intervals of 0.001. For each *p*, 1000 runs were made and the average is being used. My code ran in about 45s in my system.

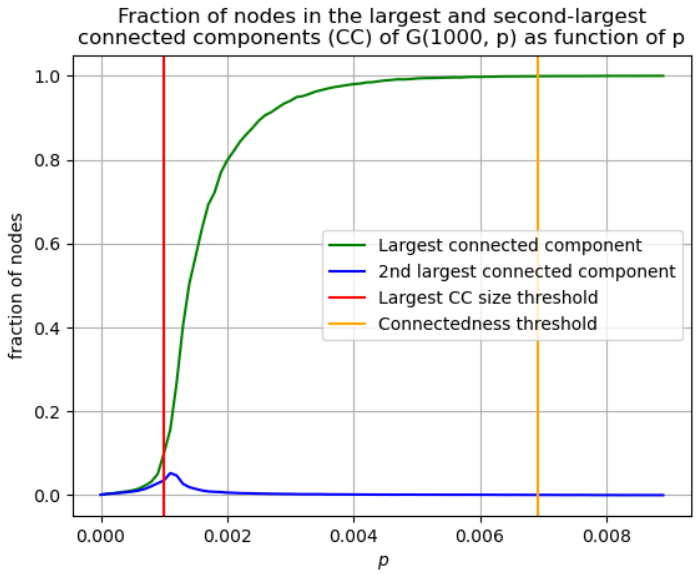
`tqdm.tqdm()` is used in this and subsequent problems to visually show how much computation is completed and is left through a moving progress bar.

Question 4

A method `oneTwoCompomentSizes` is implemented in the class `UndirectedGraph`. It calls a helper function `__bfs_size()` that performs BFS and returns the size of connected component the starting vertex belong to.

A function `verify_er_one_two_connectivity()` is defined that verifies the statement *"If p<0.001, the Erdős-Rényi random graph G(1000,p) will almost surely have only small connected components. On the other hand, if p>0.001, almost surely, there will be a single giant component containing a positive fraction of the vertices."*

Here also, the function is specific to n=1000. The computation is done for *p* at interval of 0.0001 from 0 to 0.0089. The above-implemented method `oneTwoCompomentSizes` is used to get the size of top 2 largest connected components. This is plotted and displayed. My code ran in about 3.5min in my system.

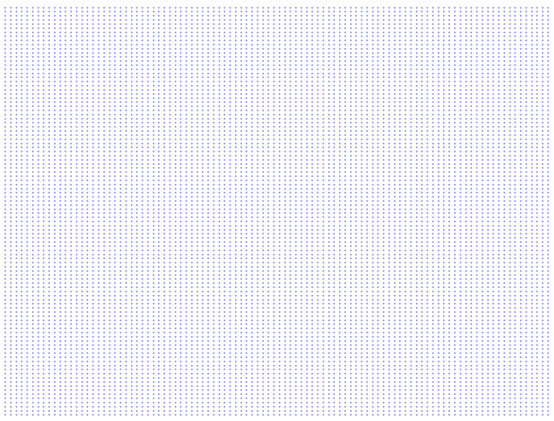


Output for verification part

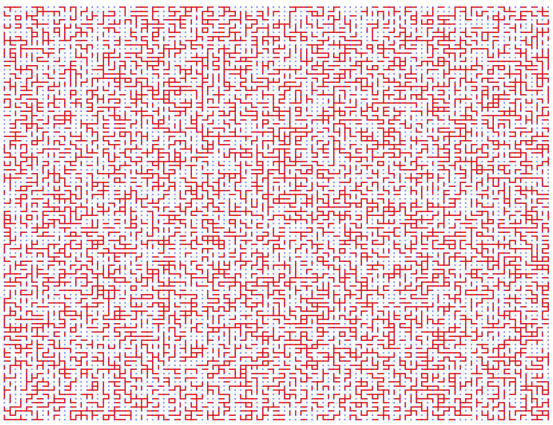
Question 5

Class Lattice is made, such that a mandatory argument of non negative integer n should be passed to the constructor and a 2D $n \times n$ lattice is made. Function `empty_graph()` from `networkx` was used and nodes were added using the function `add_node()`. Method `show_graph` was implemented which displays the graph. Method `percolate` adds all possible edges for a 2D lattice, and later removes edges based on sampling outcome. This was done because adding all edges at once and removing unwanted edges was found to be easier to implement than adding necessary edges during sampling.

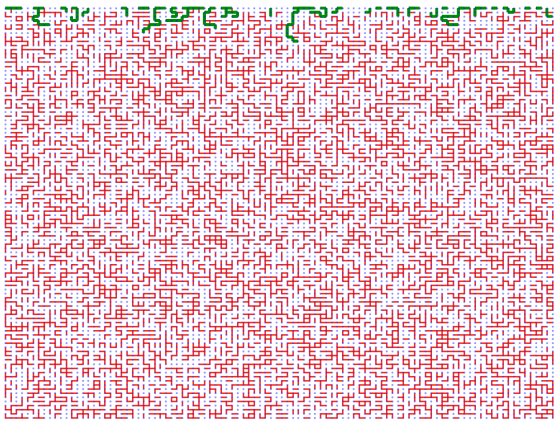
Method `existsTopDownPath` was implemented which checks if a path exists between any node in top layer and any node in the bottom layer using the function `has_path()` from `networkx`. Method `showPaths` was implemented which, for each top layer vertex, displays (case 1) the shortest path to any bottom layer vertex, if any exists, and (case 2) the largest shortest path from that vertex, otherwise. Function `shortest_path()` from `networkx` was used to get shortest path between any 2 vertices, and this was used to handle case 1, where some top-down path exists. Shortest one of all paths to bottom layer from each vertex is plotted in this case. Otherwise, a function from `networkx`, `single_source_dijkstra_path_length()` is used to compute the shortest path lengths from the source to all reachable vertices, and longest of all such ones are used to plot the path for case 2. The function `shortest_path()` takes about 2min to complete in my system for n=100, p=0.7.



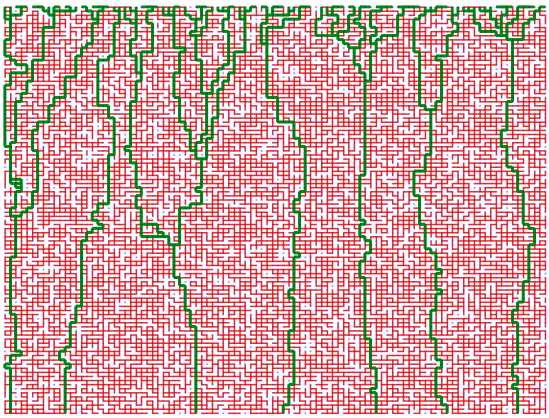
Lattice with n=100



Percolated lattice with n=100 with p=0.4



Output of `showpaths` on the above right lattice



Output of `showpaths` on lattice with
n=100 percolated with p=0.7

Question 6

We use the above-implemented method `existsTopDownPath` to implement the function to verify the statement given in the problem,
"A path exists (almost surely) from the top-most layer to the bottom-most layer of a 100 × 100 grid graph only if the bond percolation probability exceeds 0.5."

Here also, the function is specific to n=100. Values of fraction of top-bottom percolation occurred is computed for p from 0 to 1 in intervals of 0.01. It took about 11 min for this to complete in my system.