Computer Science 2XC3 - Graded Lab II

It is increasingly becoming easy to generate automated solutions to popular problem; partly because of AI tools, but also because of vast online communities that exist to find new solutions and improve exiting solutions to popular problems. Whether or not that solution is correct and applicable to our context, can be assessed only if we understand the concepts and can critically evaluate them. The goal of this lab is to motivate you to not only produce the correct solution to problems, but also to reflect on why, how and when the solution is likely to succeed or fail.

Instructions:

- Complete all work using python programming language.
- Please read all problem descriptions carefully before beginning the lab work.
- Feel free to seek the help of TA's if you need clarifications on the task.
- Please do not hard code any results.
- Your submission must include a *.py file and a *.pdf file on Avenue.
- Please name your files as <**mac_id**>_Lab_II_code.py and <**mac_id**>_Lab_II_report.pdf.
- The report must be limited to maximum 5 pages, 12-point Times New Roman font size. All charts must be included in the report.
- The lab has 7 parts.

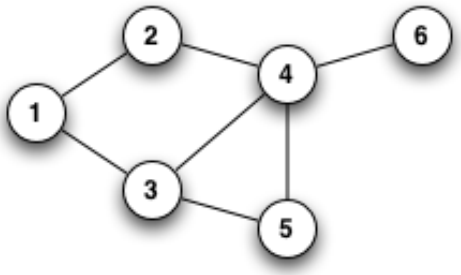
Grading Rubric: This graded lab is total 50 points

Algorithm	Max Points
Part 1	5 points
Part 2	5 points
Part 3	5 points
Part 4	5 points
Part 5	5 points
Part 6	9 points
Part 7	5 points
Questions are answered correctly	6 points
Code is well organized and commented	5 points

In class, we discussed a simple implementation of the undirected and non-weighted version of the graph. We have discussed hash map and adjacency list representations of the graph and traversing the graph using BFS and DFS. When using **BFS** and **DFS**, we mostly returned the visited nodes.

Part 1: In this part, implement two functions, **BFS_2** and **DFS_2** which return the path between two nodes. The functions take input arguments as graph, source and destination, and return a list of nodes in sequence of visiting. For instance, in a graph G, if to reach node 8 from node 6, one needs to traverse the path starting at 6 to 23, to 12, then to 5, then to 10, and finally to 8, your function BFS_2 (G, 6,8) should return a list [6,23,12,5,10,8]. Please note that the list returned by DFS_2 (G, 6,8) may be different.

Part 2: In some applications, we need to find connections from a given node to all nodes. Think about how one might find recommendations for possible connections on social media platforms. In this variation implement **BFS_3** and **DFS_3** which takes 1 node as an input and return paths to every other node from that node (please note that this is different from all paths between all nodes. Your goal is to find a path to all nodes from a given node). These paths should be returned as a "predecessor dictionary". Predecessor dictionary contains the key as the current node and the value as the predecessor node. For example, for the following graph G, your implementation of BFS_3 (G, 1) will return the predecessor dictionary as: {2: 1, 3: 1, 4: 2, 5: 3, 6: 4}. Predecessor dictionary allows us to reconstruct the path from starting nodes by simply connecting the predecessors of each node backward. So, the path from 1 to 6 is $1 \rightarrow 2 \rightarrow 4 \rightarrow 6$.



If there is no path between 2 nodes there will be no entry in the dictionary corresponding to that.

Part 3: Implement the following two functions in the graph class:

- **has_cycle( )** that computes and returns True if the graph has a cycle.
- **is_connected( )** that computes if there is a path between two nodes and returns True if the path exists. Please note that this is different from what we discussed in class (**has_edge( )**). While has_edge finds whether an **edge** exists between two nodes, **is_connected** finds whether there is a **path** between two nodes.

Part 4: In the previous lab we conducted a few experiments using a random list generator that I provided. What would that look like for a graph? To experiment with graphs, you want to be able to generate random graphs. Write a function to do so. The way to approach this is to think about the essential elements of the graph; nodes (n) and edges (e). So when you call the function create_random_graph(n,e), it should create a random layout with only a single edge between two

nodes. You may use any approach to represent your generated graphs, depending upon the implementation you are creating.

Part 5: In this part, design an experiment to compute the probability of a graph having a cycle, when you generate a random graph with n nodes and e edges. This is an open-ended question so think about how you would design the experiment. One possibility is as below:

1. Create 100 graphs using a fixed number of nodes (or edges).
2. Compute for each graph, whether it has a cycle (you have already written this function in the previous section).
3. Then compute the proportion of the random graphs that had cycles.
4. Run steps 1-3 for several iterations.

In your report, please include the number of iterations you ran, and the results you got. You can run this experiment multiple times and your graph should show the probability computed in each iteration. Also report the average proportion across all trials.

Part 6: Computing minimum vertex cover is a basic combinatorial optimization problem where the goal is to determine a minimum subset of vertices that cover all edges. The python code given contains a function to compute the minimum vertex cover for an undirected graph. It works for graphs for small node sizes (<30). Implement the below 3 variations of approximation algorithms for the Vertex Cover Problem. All functions must return the list of nodes that form the minimum vertex cover.

a. Function mvc_1 (G) takes in an object of Graph (g) and does the following:
   1. Start with an empty set C = {}
   2. Find the vertex with the highest degree in G, call this vertex v.
   3. Add v to C.
   4. Remove all edges incident to node v from G.
   5. If C is a Vertex Cover return C, else go to Step 2.

b. Function mvc_2 (G) takes as an input, an object of Graph (g) and does the following:
   1. Start with an empty set C = {}
   2. Select a vertex randomly from G which is not already in C, call this vertex v
   3. Add v to C
   4. If C is a Vertex Cover return C, else go to Step 2

c. Function mvc_3 (G) takes as an input, an object of Graph (g) and does the following:
   1. Start with an empty set C = {}
   2. Select an edge randomly from G, call this edge (u, v).
   3. Add u and v to C.
   4. Remove all edges incident to u or v from G.
   5. If C is a Vertex Cover return C, else go to Step 2.

Note: When you remove (or perform any other operation on) an edge, do not directly manipulate the graph, instead work on a local copy of the input graph.

Part 7: Evaluate whether (or not) the above algorithms (6.a, 6.b and 6.c) return the minimum vertex covers, by using a suitable experiment design. One possible experiment design is as below:

a. Generate 100 random graphs with 6 nodes and e edges where e = {1,5,10,15,20}. You can use the function you implemented in Part 5 for this.
b. Find the minimum Vertex Cover (MVC) for each graph and sum the size of all these MVCs (this can be a potential baseline).
c. Run each of your approximations (6.a, 6.b, and 6.c) on each of the same 100 graphs (**do not modify the graphs within your experiment**). Keep track of the sum of the sizes over 100 graphs of each approximation's Vertex Covers.
d. Then you can measure an approximation's expected performance by looking at that approximation's size sum (computed in 7.c) over the sum of all MVCs (computed in 7.a)
e. Graph each of the approximation's **expected performance** (computed in 7.d) as it relates to the number of edges on the graph (set e indicated in 7.a)

In total, you should have **at least** three (3) meaningful graphs in this section. This does not mean 1 graph for each approximation! You should be able to plot each of the approximation's curves on a single graph.

In the report answer the following questions:

a. For a given random graph with n nodes and edges, what proportion of the minimum vertex cover is expected from mvc_1, mvc_2, and mvc_3? You may answer the question using a table.
b. Is there a relationship between how good we would expect an approximation to be and the number of edges in a graph? In general, does the approximation get better/worse as the number of edges increases/decreases?
c. Is there a relationship between how good we would expect an approximation to be and the number of nodes in a graph? In general, does the approximation get better/worse as the number of nodes increases/decreases? To answer this question, you may have to run an experiment where you vary the number of nodes instead of edges. Include the code of this experiment in your submission.
d. Remember to include details of Part 5.

Your report must be a **maximum of 5 pages**, Times New Roman, 12-point font size. This format is mandatory for all submissions.