Computer Science 2XC3: Graded Lab III

Please read all instructions carefully. If you have any questions please reach out to your TA's to guide you through the task. All the questions are provided in this pdf and the code is in the ipynb file. You have the freedom to incorporate your code and reflections all in 1 ipynb, or code in notebook and reflections (and charts) in separate pdf. You will submit a ipynb file, a pdf-converted version of your .pynb file (and your reflections file in case you decide it to be separate). Whatever you decide, keep your work organized. Answers that are difficult to locate in a maze of text and code will **not** be graded.

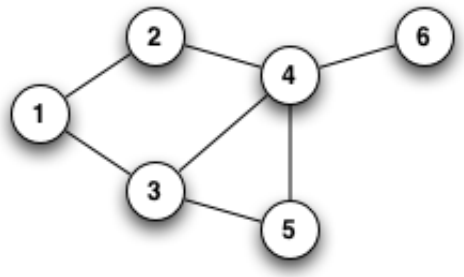
Part 1

In class, we discussed a simple implementation of the undirected and non-weighted version of the graph. We also talked about the directed graph implementation where both the path between "src" and "dst" nodes is stored. We will also discuss how to search if a node exists in a graph using Breadth First Search (BFS) and Depth First Search (DFS). Using these techniques, the functions we discussed return True if the path exists between 2 nodes, otherwise it returns false.

This implementation is limiting since not much can be accomplished with True and False. In this lab, implement two variations of BFS and DFS.

Part 1.1: Implement BFS2 and DFS2 where the path between two nodes node1 and node2 is returned as a list. For instance, in a graph, 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 BFS2(graph, 6,8) (or DFS2(graph, 6,8)) should return a list [6,23,12,5,10,8]. Implement both **BFS2** and **DFS2** for this variation.

Part 1.2*:* 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 BFS3 and DFS3 which take as an input 1 node and return paths to every other node (note that this is different from all paths between all nodes. Your goal is to find **a path to a node**). These paths should be returned as a "predecessor dictionary". Predecessor dictionary contains the key as the node and the value as the predecessor node. For example, for the following graph, your implementation of BFS3(graph, 1) will return the predecessor dictionary as: {2 : 1, 3 : 1, 4 : 2, 5 : 3, 6 : 4}



Now it is possible to reconstruct the path from this dictionary by simply connecting the predecessors of each node backward. So the path from 1 to 6 is 1 -> 2 -> 4 -> 6. If there is no path between 2 nodes there will be no entry in the dictionary corresponding to that.

Part 1.3*:* Implement a function in the graph class called **has_cycle( )** that computes and returns True if the graph has a cycle.

Part 1.4: Implement a function in graph class called **is_connected( )** that computes and returns True if there is a path between two nodes. 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.

Feel free to use any part of the code here.

Part 1.5*:* 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.

Part 1.6: By now, you should have a good understanding of how to design an experiment to answer a question. 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.

HINT: Create a sufficiently large number of graphs using a fixed number of nodes (or edges??). compute for each graph, whether or not it has a cycle (you have already written this function in the previous section). Then calculate what proportion of the random graphs had cycles.

As usual, there are multiple ways you can approach this problem, so be creative. In your reflection, describe your experiment design, the number of iterations you ran, why you chose a specific experiment design etc. You can run this experiment multiple times and your graph should show the probability computed in each iteration.

Part 1.7: Similar to Part 1.6 design an experiment to compute the probability of a graph being connected when you generate a random graph with n nodes and e edges. Again, you can use a similar strategy and describe in your reflection, your experiment design, and number of experiments and justify your choices.

Part 2

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 jupyter notebook contains a function to compute the minimum vertex cover for an undirected graph. It works for graphs for small node sizes (<30). Implement 3 different approximation algorithms for the Vertex Cover Problem.

Part 2.1: Approx1 (graph) takes in an object of Graph 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

Part 2.2: Approx2(graph) takes as an input, an object of Graph 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

Part 2.3: Approx3(graph) takes as an input, an object of Graph 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 2.4: Evaluate whether (or not) the above algorithms return the minimum vertex covers. In case they do not compute how far off the minimum is? For a given random graph with n nodes and e edges, what proportion of the minimum vertex cover is expected from approx1, approx2, approx3?

One important thing that you would need is what is the "actual" minimum vertex cover against which you are comparing, what is the baseline. Be creative here.

HINT: A potential experiment can be:

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

In total, you should have at least three 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. Some questions to consider in your experiment:

- 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?
- 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?
- The approach described in the Potential Experiment is getting at the average performance of the approximation. What about the worst case of the approximation? To figure that out we would have to test our approximations on every single graph for approx1(). And for the other two the non-deterministic nature of the algorithms makes this even more problematic. However, we may be able to test the worst case for approx1() on very small graphs. How would you generate all graphs of size 5 for example?

Discuss these in your reflection section.

Part 2.5: Another graph problem is the Independent Set problem. Given a graph G = (V, E) we say S is an Independent Set in G if and only if:

$$S \subseteq V \text{ and } \forall u, v \in S, (u, v) \notin E$$

Or S is an Independent Set if there are no edges in G connecting the nodes in S. In general, it is easy to find an Independent Set of G. For example, {} is trivially an Independent Set. It is much harder to find the largest Independent Set in a graph G.

Implement a function MIS(G), which returns a maximum Independent Set in G. Hint, brute force this similar to how we brute forced the MVC problem. But you can use some logic if you want are brave.

Part 2.6: Experiment with some random graphs and MIS and MVC. Is there a relationship between the minimum Vertex Cover and the Maximum Independent Set?

Hint: yes. Determine what this relationship is. To get started, generate some random graphs with n nodes. When you sum the size of the MIS and the size of the MVC, what do you observe?

Part 2.7: Inspect the MIS and MVC directly as well. What can you empirically conclude? Provide your observations in the reflection section.