Computer Science 2XC3: Lab 2

Please read the following carefully. There are no completion/participation grades in labs. However, the TAs are there to guide and aid you in your tasks.

Purpose

This lab focuses on implementing graphs and graph algorithms. Furthermore, you will conduct experiments to gain a deeper insight into the properties certain graphs hold, and the corresponding likelihood (probability) of properties holding. This lab will span two weeks. In the first week you will cover the following (but not necessarily limited to):

- Reimplement BFS and DFS to return paths instead of Booleans
- Implementations of functions to determine if a graph has a cycle and/or is connected
- A strategy and implementation to create random graphs
- Experiments to show the probability of graphs having certain properties

During the second week you will cover the following (but not necessarily limited to):

- Investigate the classic Vertex Cover and Independent Set problems
- Implement brute force algorithms for these two problems
- Design and implement several approximation algorithms for these problems
- Run experiments to gauge how “good” of an approximation your algorithms are
    - Can you say you are never greater(lesser) than some percentage of the minimum(maximum)?
    - Can you say with a high probability your algorithm returns an answer within some factor of the minimum/maximum?

Part 1

Throughout this lab you will be creating and ultimately submitting a lab report. Your lab report should look professional and complete. It should include the following:

- Title page
- Table of Content
- Table of Figures
- An executive summary highlighting some of the main takeaways of your experiments (this can be presented in bullet form if you wish)
- A clearly marked section for each experiment/exercise outlined in this lab
- An appendix explaining to the TA how to navigate your code (for example, which .py file to find which implementation/experiment in)

For each experiment, include a clear section in your lab report which pertains to that experiment.

BFS and DFS

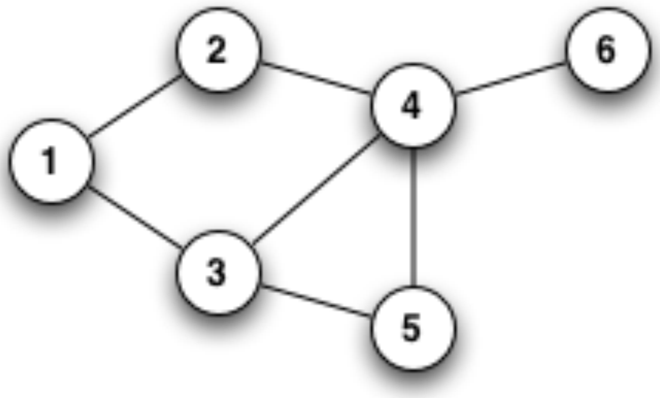
In *graphs.py* you will find an implementation of an undirected (non-weighted) graph alongside some simplistic implementations of Bread First Search (BFS) and Depth First Search (DFS). If everything goes according to plan, we will also be going over these elements in lecture as well. Right now, BFS and DFS are not the most useful implementations. They simply return True if and only if a path exists from *node1* to *node2* (using the Breadth and Depth approaches).

Implement variations on these which return the path from *node1* to *node2* as a list of nodes, starting with *node1* and ending with *node2*. For example, BFS2(G, 3, 5) would return:

[3, 10, 4, 1, 5]

if via BFS a path from 3 to 5 was found by first going to 10, then 4, then 1, then finally 5. If no path is found the function should return an empty list. Name these functions **BFS2** and **DFS2** and include them in your graphs.py file.

Now implement variations on BFS and DFS which do not take in a second node. Instead, the function finds a path to every node (which there is a path to) and returns these paths in the form of a ``predecessor dictionary". Name these functions **BFS3** and **DFS3** and include them in your graphs.py file. This predecessor dictionary encodes all the path information. For example, for the graph below:



the BFS3(G, 1) predecessor dictionary would be:

{2 : 1, 3 : 1, 4 : 2, 5 : 3, 6 : 4}

Note, from the above one could build the path from 1 to 6 by working backwards from 6 looking at its predecessors until they reached 1. If there is not path to a node *k* from the input node, then *k* should not appear in the predecessor dictionary.

For the following two functions, reuse your BFS and/or DFS code where possible.

Implement a **has_cycle(G)** function in your graph.py file. This function should return True if and only if there is a cycle in G.

\Implement a **is_connected(G)** function in your graph.py file. This function should return True if and only if there is a path between any two nodes in G.

Experiment 1

Before we can run experiments, we need to figure out a way to randomly generate graphs – similar to how we randomly generated lists in the last lab. It is not too complicated, but may take a bit more thought than the lists. Implement a function create_random_graph(i, j), which returns a graph with I nodes and j edges. Note, your graph should **not** create a graph with "multiples" of the same edge.

For this experiment we want to be able to confidently answer questions like the following:

<u>If I have an arbitrary graph with *i* nodes and *j* edges, what is the probability the graph has a cycle?</u>

I want to give you some relative freedom with this experiment to determine how to answer this question, but if you are having trouble getting started consider the following. Fix the number of nodes in your graph, for example, 100. Then for each "number of edges" value which makes sense create m random graphs. What proportion of those has a cycle? In your report this section should include:

- An explicit outline of the experiments you ran. For example, number of node, number of edges, runs, etc.
- At least one graph of *the number of edges vs cycle probability* displaying the results of your experiments.

- Note, you can change this to *proportion of edges* instead of *number* if you wish; this would then allow you to display multiple curves representing multiple values of the number of nodes on the same graph

- A brief discussion and conclusion regarding the results. A few sentences are fine here.

Experiment 2

Similar to Experiment 1, for this experiment we want to be able to confidently answer questions like the following:

<u>If I have an arbitrary graph with *i* nodes and *j* edges, what is the probability the graph is connected?</u>

Use a similar strategy to testing this that you did in Experiment 1. That is, fix the number of nodes in your graph. Then for each "number of edges" value which makes sense (feel free to skip some values if you want) create m random graphs. What proportion of those *m* graphs are connected? In your report this section should include:

- An explicit outline of the experiments you ran. For example, number of node, number of edges, runs, etc.

- At least one graph of *the number of edges vs connected probability* displaying the results of your experiments.

    - Note, you can change this to *proportion of edges* instead of *number* if you wish; this would then allow you to display multiple curves representing multiple values of the number of nodes on the same graph

- A brief discussion and conclusion regarding the results. A few sentences are fine here.

Part 2

I have added some code to the graph.py file which will allow you to find the minimum vertex cover of an undirected graph. Note this algorithm is exponential and will not work for graph of even a moderate size. More than 30 nodes is pushing it.

Approximations

You will implement three different approximation algorithms for the Vertex Cover problem. Name your three algorithms approx1(G), approx2(G), approx3(G), respectively. Each takes in a single graph as input. This graph is an object of the class Graph in graph.py. Find a description of each approximation below. Note: When I say things like remove an edge from G, you should not modify G directly. Make an equivalent local copy of G if that helps.

approx1(G)

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

approx2(G)

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

approx3(G)

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

Approximation Experiments

The above three algorithms may not necessarily return minimum vertex covers (although they may). But if the Vertex Cover they return is not minimal, how far off the minimum is it? Or phrased differently, if I were to select a graph at random with n nodes and m edges, what percentages of the minimum vertex cover would I expect each approximation to return? You will run some experiments to answer these questions with a reasonable level of certainty.

In order to run these experiments we must be able to determine the size of the minimum vertex cover to compare it against. Therefore, we will be limited to relatively small graphs. I invite you to run any interesting experiment you can devise. But if you are having trouble getting started, consider the following.

Potential experiment:

- Generate 1000 random graphs with 8 nodes and m edges where m = 1,5,10,…,30
- Find the minimum Vertex Cover (MVC) for each graph, sum the size of all these MVCs
- Run each of your approximations on each of the same 1000 graphs (careful, your approximations should not be modifying the graphs directly!). 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:

- 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 really 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?

The Independent set Problem

Another graph problem you likely not have heard of 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 if you skipped your discrete math classes, 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.

Experiment with some random graph 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? Inspect the MIS and MVC directly as well. What can you empirically conclude?

Grading and Submission

Your group will submit the following documents to Avenue:

- report.docx (or .pdf, or whatever – as long as a reasonable person can open it)
- code.zip (all your source code pertaining to the lab – including experiment code)

In addition to the grade allocations below, your report may lose up to 20% of the final grade for not looking professional, having formatting/style issues, graphs presented in a messy manner, etc. Moreover, you may lose grades for not including elements explicitly mentioned in the Part 1 section of this document. Find a rough grade breakdown below:

Part 1

BFS and DFS implementations	20%
Connected and Cycle implementations	10%
Experiment 1	10%
Experiment 2	10%

Part 2

VC Approximations	20%
Approximation Experiments	20%
IS vs VC Analysis and Discussion	10%