COMPSCI-2XC3 — Lab 8

DUE : Sunday, March 20, 2022 11:59 PM EST

Submit the lab on Avenue. Note, Avenue must receive your lab by the due date. **Do not leave your submission to the last minute! Late submissions, even by seconds, will not be graded.** You have been warned.

This lab is worth (59/7)% of your final grade in the course. Read this document carefully and completely. **Whenever I ask you to discuss something, I implicitly mean to include that discussion in your lab report. Furthermore, when I ask you to run an experiment or evaluate the performance of a function/method include scatter plots where appropriate. Do not import external libraries, especially those containing data structures you are meant to implement!** Include all timing experiments in your `code.py` file.

Purpose

The goals of this lab are as follows:

1. Understand minimum spanning trees (MSTs).
2. Implement different variations on Prim's algorithm.
3. Intelligently use heaps/priority queues in graphing algorithms

Submission

You will submit your lab via Avenue. You will submit your lab as three separate files:

- code.py
- mst.py
- report.pdf (or docx, etc.)

Only one member of your lab group will submit to Avenue – see the section below for more details on that. Your report .pdf will contain all information regarding your group members, i.e. name, students number, McMaster email, and enrolled lab section. This will be given on the title page of the report. For the remainder of this document, read carefully to gauge what other

material is required in the report. Your report should be professional and free from egregious grammar, spelling, and formatting errors. Moreover, all graphs/figures in your report should be professional and clear. You may lose grades if this is not the case. Organize the report in a such a way to make things easy for the TA to grade. You are not doing yourself any favors by making your report difficult to mark!

Testing

When we test your MST algorithms you may assume:

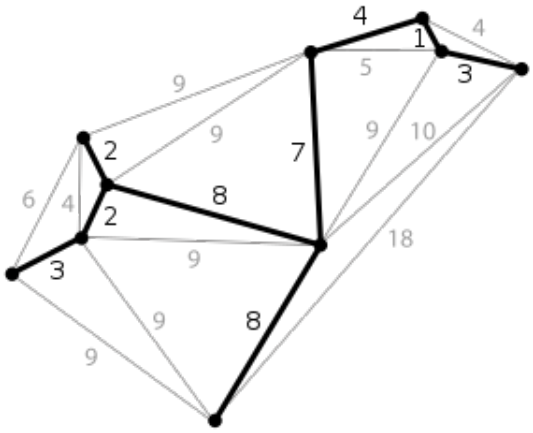
- The input graph is connected
- The edges weights are all positive and less than 1000.
- All edge weights are unique. That is, no two edges have the same weight.

It may interest you to know that if a graph has unique edges weights, it also has a unique MST.

Prim's Algorithm Version 1 [35%]

In `lab8.py` you will find an implementation of an undirected weighted graph. Familiarize yourself with this implementation. This is the class we will be using to test your functions. I will also give as brief overview of it in lecture.

You should have learned minimum spanning trees (MSTs) in 2C03 course. Otherwise also, they are actually quite intuitive. Given a weighted graph, a spanning tree of that graph is a subset of edges from the original graph such that it is acyclic and connected (aka a tree). A MST is a spanning tree such that the sum of all the edge weights in the MST is minimized. Informally, you are trying to connect the graph using the fewest number and least weighted edges as possible. Below shows a graph and a MST of that graph bolded within it.



Prim's algorithm follows this basic strategy: given a graph $G = (V, E)$, create an MST by starting with an arbitrary node and putting that node in some set A. Then the tree is constructed by

1. Find the least weighted edge (v_1, v_2) such that, one of v_1 or v_2 is in A, and the other is not.
2. Add that edge to your MST, and add the node which was not in A, to A.
3. Repeat until $A = V$, that is, all the nodes have been connected.

It is simplistic on the surface, but how you find and update least weighted edges can have a large impact on the performance of the algorithm.

For your first implementation, do not worry about efficiency. Just make it work. In your `mst.py` file write a function `prim1()` which takes in a WeightedGraph as input. Is returns a WeightedGraph as output which will represent an MST. To find the least weighted edge which meets the criteria described above just iterate through the graphs adjacency list.

Prim's Algorithm Version 2 [50%]

Posted with this lab is a minimum heap implementation. See `min_heap.py`. This differs from our previously seen max heap implementation by:

- Implementing a decrease key method
- Elements of the heap have a key and a value attribute
- To achieve an efficient decrease key method the heap also contains a map which keeps track of which index each value is stored.

I will be reviewing the implementation of this heap in lecture as well.

Here is the idea, you will use the min heap to keep track of the current least weighted edge which connects a new node to A. The Elements of your heap will have the node as the value, and the key will be the least weighted edge connecting that node to A. Initialize your heap by adding all the nodes with ∞ as the key. Choose an arbitrary node to start with (remove that from the heap or don't add it to begin with). Loop through that node's adjacent nodes, and update (decrease) all the keys to the edge weights that connect them. Extract the min, add the appropriate edge to the MST, and repeat the process of decreasing keys of the node you just extracted.

Do not modify the MinHeap or Element class. Name your implementation `prim2()` in your `mst.py` file. Your implementation should run in $O(E \log E)$ time. You do **not** need to verify this empirically.

Prim vs Prim [15%]

It should not be a surprise that Prim v2 is more efficient than v1. But how much more efficient is it? Design an experiment to highlight the gain in performance v2 offers. I am leaving this pretty open ended, in your report present some empirical results and comment on the advantages of v2.