CCPS616 Lab 3 – Minimum Spanning Trees

Preamble

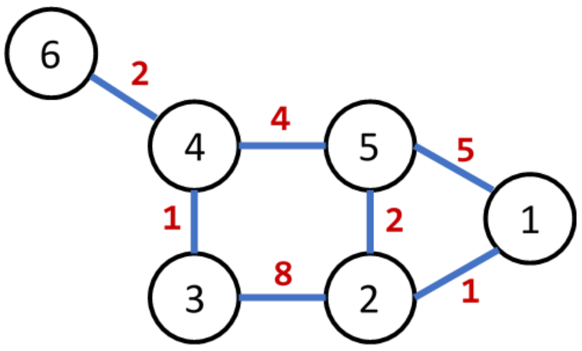
In this lab you will write a C++ program that finds a minimum spanning tree of a graph. Your implementation must be from the ground up without linking any external libraries. Your lab should be completed in a single cpp file called **lab3.cpp**, and should compile at the command line very simply as follows:

```
g++ -o lab3 lab3.cpp
```

Please do not submit labs that don't even compile. We're past that. I'm not going to fix your syntax errors. If you can't make any progress on the lab whatsoever, submit a simple program that prints a message saying as much. For this, I will award you a single mark.

Lab Description

In this lab you will write a program that accepts a graph as input and produces a *minimum spanning tree* as output. The input graph will be read from a plain text file and will come in the form of an adjacency matrix. An example of the input and associated weighted graph can be seen below.



```
6
0 1 0 0 5 0
1 0 8 0 2 0
0 8 0 1 0 0
0 0 1 0 4 2
5 2 0 4 0 0
0 0 0 2 0 0
```

In the above example, the first number (6) in the file indicates the number of vertices. The values below form the actual adjacency matrix. The values in the matrix indicate the weight of the edge between two vertices. If the value is zero, there is no edge. You may assume the input graph is simple, finite, and connected.

You may choose either Prim's or Kruskal's, whichever you prefer. In either case, the input format is the same. Note that you are not required to use an adjacency matrix internally. This is just how the graph will be encoded in the input file. It's probably a good idea to build an adjacency list when you read in the graph.

Your program will output an adjacency matrix in the exact same format as the input file. The output will contain only those edges (and their weights) that are part of the minimum spanning tree.

You are not required to implement a priority queue or disjoint set data structure to optimize the asymptotic complexity of your implementation. That is, an $O(|V|^2)$ solution is just fine. Of course, implementing the appropriate data structures to push the complexity down to $O(|E|\ln(|V|))$ is great practice, and highly encouraged.

Testing

When running your program, the graph should be read in as an input argument. Running your program should be done as follows:

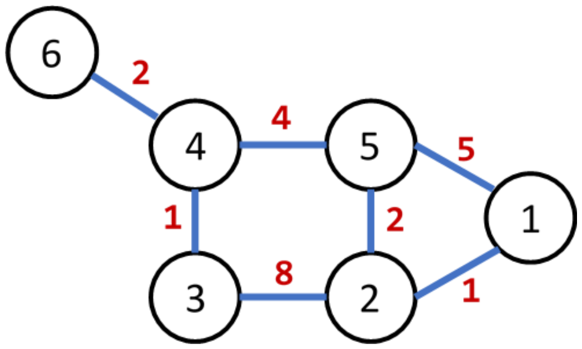
```
Linux: ./lab3 graph.txt     Windows: ./lab3.exe graph.txt
```

Your program should produce an output file called graphMST.txt. In this lab, you are marked primarily for producing a correct MST. Thus, you need not perform numerous trials and timings – simply compute and output the MST.

You should test your program on several graphs of varying sizes. I don't expect you to produce a 1000x1000 adjacency matrix by hand, but you should test on graphs of similar order as those we looked at in class. In fact, some good test cases would be exactly those graphs we saw in class. This way, you can verify the correctness or your results with the result obtained for the in-class examples.

Results

As mentioned, your results should be produced in the form of an adjacency matrix. For the same graph seen in the description, the MST would be written out to a plain text file as follows:



```
 6
0 1 0 0 0 0
1 0 0 0 2 0
0 0 0 1 0 0
0 0 1 0 4 2
0 2 0 4 0 0
0 0 0 2 0 0
```

Submission

Lab 3 may be done in **pairs**! Submit your source file (**lab3.cpp**), a sample graph file, and the MST produced for that input to D2L. If you worked in pairs, Indicate the names of both partners in a comment at the top of your source code.

Marking Rubric: This lab is simplistic, and is out of 5 marks total

3 marks – Algorithm correctness. Produces correct results on my test cases.
1 mark – Graph input is parsed properly and stored in an adjacency list.
1 mark – Output MST is produced and formatted correctly.

3 marks (bonus) – implement the min-heap in the case of Prim's, or the disjoint set in the case of Kruskal's. Get the complexity down to $O(|E|\ln(|V|))$.