ECE250 Project 4: Graphs

Due: Friday, March 24st at 11pm to the DropBox on Learn (**Extended Deadline: Friday, March 31**)

Overview

In this project, you must design and implement a graph data structure to store a weighted, undirected graph. In addition, you will design and implement an efficient algorithm to find a Minimum Spanning Tree (MST) of the graph. You must implement this algorithm in, at worst, O(|E|log(|V|)) time, where |E| is the number of edges in the graph and |V| is the number of vertices.

Datasets

We have provided some datasets for your use and a python script to generate more. A video has been uploaded to Learn to show you how to read the datasets and use the python script. These datasets represent power stations, which are numbered from 1 to N, and the cost of connecting those power stations to neighbouring stations. The datasets all have the same format as following:

- On the first line is N, the number of power stations in the dataset.
- On next lines are edges between power stations, one edge per line. These edges are given in a space-separated values format like: `i j W` where `i` is the number of the first power station, `j` is the number of the second, and `W` is the weight between them representing the cost of connecting the two stations. You may assume that `i`, `j`, and `W` are all integers greater than 0 (so 0 is not included). You may assume that the "int" data type is sufficient to store them.

By computing an MST on this graph, your design and implementation produce a minimum-cost power grid.

Program Design and Documentation

You must use proper Object-Oriented design principles to implement your solution. You will create a design using classes which have appropriately private/protected data members and appropriately public services (member functions). It is not acceptable to simply make all data members public. **Structs are not permitted.** You may notice that you are writing a generic graph implementation for this project. A Tree is a graph without a cycle. Therefore, it may be a good idea to use a single graph class to represent both the power grid and the MST. This is not a requirement, but it is a nice design decision to simplify your code.

Write a short description of your design to be submitted along with your C++ solution files for marking according to the template posted to Learn. **You may use the following libraries from the STL and all of their associated functionality:** vector and tuple.

Input/Output Requirements

In this project, you must create a test program that must read commands from standard input and write the output to standard output. The commands are described below.

Note: In order for the MST to be unique, the graph must be connected with unique edge weights. You may assume that the LOAD command always produces such a graph. However, the INSERT command may not, and the DELETE command may change the graph so that the unique MST conditions are not met. The MST command below will only be called on connected graphs with unique weights.

Command	Parameters	Description	Expected Runtime (worst case)	Output
LOAD	filename	Load a dataset into a graph. This command may not be present in all input files. If it is, you may assume that it is the first command in the input file. You may assume there are no illegal arguments in the datasets. You may assume that, if this command is used, it will be used only once per input file.	**N/A – this depends on your implementation and the dataset size. Do not analyze this function.**	**success** **Note:** This command should output "success" no matter what. It should not output one "success" per vertex.

Command	Parameters	Description	Expected Runtime (worst case)	Output
INSERT	*a b w*	Insert a new edge into the graph from vertex *a* to vertex *b* with weight *w*. If either *a* or *b* are not in the graph, add them. Valid values *a* and *b* will be positive integers between 1 and 50000 inclusive and valid values of *w* will be a positive integer weight greater than 0	**Depends on your implementation. Analysis is expected. See below.**	**success** if the insertion was successful **failure** if the edge is already in the graph, regardless of weight **illegal argument** (see below the table)
PRINT	*a*	Print all vertices adjacent to vertex *a*. Valid values for *a* will be a positive integer between 1 and 50000. The order in which you print the vertices is not important. The Autograder will be programmed to handle any ordering.	**O(degree(a))**	**b c d …** Print all vertex numbers adjacent to *a* on a single line with spaces between them, followed by a newline character. **failure** if vertex *a* is not in the graph **illegal argument** (see below the table)
DELETE	*a*	Delete the vertex *a* and any edges containing *a*. Note that this means you will need to remove the vertex *a* from the edge set of all vertices adjacent to *a*. This command may produce an unconnected graph.	**Depends on your implementation**	**success** if the vertex is in the graph and was erased **failure** if the vertex is not in the graph, including the case where the graph is empty **illegal argument** (see below the table)
MST		Find the MST of the graph. You may assume that the graph is connected. **Note:** This command must output the list of edges in the MST. The output should contain all the edges on a single line, separated by spaces. Each edge should be displayed in the format of vertex 1, vertex 2, and weight. The order in which the edges are printed is not important, as long as each edge is displayed in the correct format, the autograder will figure it out.	**O(|E|log(|V|)) to find the MST. Do not consider printing as part of this algorithm.** **|E|= number of edges in the graph** **|V|=number of vertices in the graph**	**V1 V2 W12 V3 V4 W34 V1 V4 W14…** where Vi, Vj, and Wij are the integers representing the vertices and weight of the edge. **failure** if the graph is empty.
COST		Determine the cost (the sum of the weights) of the power grid computed via the MST. You may assume the cost will never overflow an int on eceubuntu.	**O(|E|log(|V|)) (why?)**	**cost is x** where “x” is the cost of the grid. Note that x may be 0 if the graph is empty

Command	Parameters	Description	Expected Runtime (worst case)	Output
END		Last command for all input files.		This command does not print any output

Illegal arguments: For the commands *INSERT, PRINT,* and *DELETE,* you must handle invalid input. A vertex is invalid if it is larger than 50000 or less than or equal to 0. A weight is invalid if it is less than or equal to 0. No other types of invalid input will be tested. All invalid input will fit into an "int" datatype. Your code must throw an illegal_argument exception, catch it, and output **"illegal argument"** (without quotes) if it is caught. You may assume that the only type of invalid input will be integers outside of the range. To do this, you will need to:

a. Define a class for this exception, call it **illegal_exception**
b. Throw an instance of this class when the condition is encountered using this line:

```
throw illegal_exception();
```

c. Use a try/catch block to handle this exception and print the desired output of the command

You must analyze the **worst-case** runtime of your algorithms in your design document. Prove that your implementation achieves these runtimes. For vertex insertion and deletion, this runtime is dependent on your implementation. Analyze your own implementation and discuss the worst-case runtime.

Valgrind and Memory Leaks

20% of your grade in this project will be allocated to memory leaks. We will be using the Valgrind utility to do this check. The expected behaviour of Valgrind is to indicate 0 errors and 0 leaks possible, with all allocated bytes freed. To test your code with Valgrind, presuming you are using an input file test01.in, you would use the following command:

```
valgrind ./a.out < test01.in
```

Test Files

Learn contains some sample input files with the corresponding output. The files are named test01.in, test02.in and so on with their corresponding output files named test01.out etc. All test files are provided as-is. Your code must be able to parse them.

Submitting your Program

Once you have completed your solution and tested it comprehensively on your own computer or the lab computers, you should transfer your files to the eceUbuntu server and test there. We perform automated tested on this platform, so if your code works on your own computer but not on eceUbuntu it will be considered incorrect. **A makefile is required for this project since the exact source structure you use will be unique to you.** Do not submit the dataset or precompiled binaries.

Once you are done your testing you must create a compressed file in the tar.gz format, that contains:

- A typed document, maximum of four pages, describing your design. Submit this document in PDF format. The name of this file should be xxxxxxxx_design_pn.pdf where xxxxxxxx is your maximum 8-character UW user ID (for example, I would use my ID "mstachow", not my ID "mstachowsky", even though both are valid UW IDs), and n is the project number. In my case, my file would be mstachow_design_p4.pdf.
- A test *.cpp file that contains your main function that reads the commands and writes the output
- Required header files that you created.
- Any additional support files that you created. Do not submit any input files.
- A makefile, named Makefile, with commands to compiler your solution and creates an executable. Do not use the -o output flag in your makefile. The executable's name must be a.out.

The name of your compressed file should be **xxxxxxxx_p4.tar.gz**, where xxxxxxxx is your UW ID as above.