ECE 250 — Winter 2024
Electrical and Computer Engineering
University of Waterloo

Due Date: Apr 8, 2024
Professor: Ziqiang Patrick Huang
Lab Instructor: Ahmed Fahmy

Lab 4:

Emergency Response

1 Overview

The objectives of Lab 4 are:

- Implement a graph data structure that represents a weighted, undirected graph
- Implement Dijkstra's algorithm to determine the lowest-weight path between two vertices

2 Background: Emergency Response

In this project, we model a city's road network where intersections are represented by vertices and the time it takes to travel between two intersections (edges) is captured as edge weights. The primary objective for first responders is to reach their destinations in the shortest possible time. To achieve this, they must identify the route that minimizes travel time, considering three key factors: the distance between intersections, the safe speed limit on each road, and current traffic conditions. For instance, between two equally long routes, the one with a higher safe speed limit is preferred. However, should the faster route be experiencing heavy traffic, making it slower in reality, the alternative, less congested route may become the better option despite its lower speed limit. Modern cities have various means to collect real-time traffic information. For the purposes of this project, we'll integrate this traffic data into a single metric known as the "adjustment factor", which ranges from 0 to 1. The formula to calculate the travel time between any two connected vertices (intersections) is give by:

$$T = \frac{d}{S \times A}$$

where d represents the distance between the intersections in meters, S denotes the speed limit on the road linking them, measured in meters per second (m/s), and A is the adjustment factor. The adjustment factor, A, ranges from 0 to 1, where $A = 0$ indicates that the road is completely blocked, resulting in an infinite travel time.

3 Program Requirements

In this project, you must design and implement a graph data structure to store a weighted, undirected graph $G = V, E$, where we V is the set of vertices and E is the set of edges in the graph G. In addition, you will implement Dijkstra's algorithm to determine the lowest-weight path between two vertices.

Your program must read commands from standard input (e.g. `cin`) and write to standard output (e.g. `cout`). For each input, the program is expected to execute a designated action with a specified run-time requirement and provide the corresponding output message, as outlined in Table 1. **Note:** You may assume that all commands are valid.

Command	Parameters	Action	Output	Run-time
`insert`	`a b d s` **Ranges**: `a,b` $\rightarrow$ `int` $\in (0, 500000]$ `d,s` $\rightarrow$ `double` > 0	Insert a new edge into the graph from vertex `a` to vertex `b` with distance `d` and speed limit `s`, or update the edge **if** it exists. **If** either `a` or `b` are not in the graph, add them. **Assume** $A = 1$.	`success`	$O(1)$
`load`	`filename`	Load a dataset into a graph. Each line in the file will have the same format as a single `insert` command. **Assume** $A = 1$ while inserting. **Notes**: This command may not be present in all input files or may be present multiple times.	`success`	N/A
`traffic`	`a b A` **Ranges**: `a,b` $\rightarrow$ `int` $\in (0, 500000]$ `A` $\rightarrow$ `double` $\in [0, 1]$	Update the adjustment factor to `A` for the edge between vertices `a` and `b`, **if** the edge exists.	`success` OR `failure`	$O(1)$
`update`	`filename`	Update traffic data between all vertices in the file. Traffic data will be given in the same format as the `traffic` command. This operation is successful **only if** at least one edge is updated in the graph.	`success` OR `failure`	N/A
`print`	`a` **Range**: `a` $\rightarrow$ `int` $\in (0, 500000]$	Print all vertices adjacent to `a` on a single line with spaces between them, followed by a newline character. The order of vertices **is not important**.	`b c d...` OR `failure`	$O(\|V\|)$
`delete`	`a` **Range**: `a` $\rightarrow$ `int` $\in (0, 500000]$	Delete the vertex `a` and any edges containing `a`, **only if** `a` exists. **Note** you need to remove `a` from the edge set of all vertices adjacent to `a`.	`success` OR `failure`	$O(\|V\|)$
`path`	`a b` **Ranges**: `a,b` $\rightarrow$ `int` $\in (0, 500000]$	Print all vertices along the lowest-weight path between vertices a and b, **only if** a path exists. The order of vertices **does not matter**.	`a c..d b` OR `failure`	$O(\|E\|\log(\|V\|)$
`lowest`	`a b` **Ranges**: `a,b` $\rightarrow$ `int` $\in (0, 500000]$	Determine and print the weight (`x`) of the lowest-weight path (i.e. the sum of all T values), **only if** a path exists.	`x` OR `failure`	$O(\|E\|\log(\|V\|)$
`exit`	N/A	End the program	N/A	N/A

Table 1: Input/Output Requirements

Additional Notes

- The `load` command should always output `success` exactly once. If there are duplicates in the dataset, simply update the `d` and `s` values each time you encounter a duplicate.
- For the `print` and `path` commands, print vertex numbers with spaces between them. Print `failure` if the graph is empty or there is no path between the vertices or one of the vertices is not in the graph.
- For the `path` and `lowest` commands, if the input vertices do not exist, or if the lowest-weight path between them is ∞ (i.e. $A = 0$), print `failure`.
- Each output must be followed by a newline character.
- You may use the STL library in this project.
- In C++, STL's hash map is `std::unordered_map`.
- It is recommended to implement a graph as a hash map of node to hash map of node to edge (i.e. `std::unordered_map<node, std::unordered_map<node, edge>>`).
- You may assume that all commands and parameters are valid while grading.

Provided Files

You are provided with the following files: `driver.cpp` where the `main()` function is implemented, `graph.h` where your classes are *defined*, and `graph.cpp` where your classes are *implemented*. **Do not create, delete or change the name of the provided files**. In addition, there are some examples of input files along with their corresponding output files. The test files are named `test01.in`, `test02.in`, and so on, with their respective output files named `test01.out`, `test02.out`, etc.

4 GitLab Repository

We are going to use GitLab for code management and submission. The repository is created for you with the URL: `https://git.uwaterloo.ca/ece250-w24/lab4/ece250-w24-lab4-USERID`

5 Evaluation Criteria

Evaluation Focus

Output Accuracy and Performance: The primary focus of the evaluation for Lab 4 will be on the correctness of the output generated by your program and the Performance (Time complexity of the functions designed for different commands). We will use a variety of inputs and edge cases to test the accuracy of your results.

Deductions for Non-Compliance (up to -50 points): Marks will be deducted if your code does not adhere to the specified guidelines, including the prohibition of using C++ STL.

Note: It is imperative that you adhere to the prescribed utilization of a graph as outlined in the lab manual for your assignment. Failure to comply with this directive will result in a substantial deduction of points from your overall assignment score.

Additional Considerations

Code Inspection: Following the output evaluation, we will review your code to ensure it aligns with the provided notes and guidelines.

Code Comments: Include comments to explain your code logic and design choices.

Testing: Thoroughly test your program with various inputs to ensure its correctness.

No STL: Remember that the utilization of any C++ STL containers or algorithms is prohibited unless otherwise specified.

Performance Evaluation: For Lab 4, we will be checking the time complexity of functions you have designed for different commands.

Memory Leaks: We will be checking for memory leaks in Lab 4.

Use of AI Tools: If you utilized any AI tools in the development of your code, please include a comment in your source code indicating their use.

Code Compilation: Ensure that your code compiles without errors on ECE Linux servers (e.g., eceubuntu, ecetesla). We will make and test your code on these servers. If we cannot compile your code, you may receive a score of 0 for the marking.

Object-Oriented Design Principles:

- You must use proper Object-Oriented design principles in the implementation of your code.
- Create a design using classes that have appropriately private/protected member variables and appropriate public member functions.
- It is not acceptable to simply make all member variables public.
- You should consider separating the interface and implementation of class member functions by placing the declarations in a .h (header) file and the definitions in a .cpp (source code) file.
- Implement constructors for initialization and destructors for resource release in classes.