CS 271 (Fall 2024) Final Project – "You Know You Only Have 30 Minutes": Computing Shortest Paths

Instructor: Prof. Michael Chavrimootoo
Due: Wednesday, December 18, 11:59PM– No Extensions

1 Project Overview

In this project, you will be implementing a weighted graph class, a priority queue, and Dijkstra's algorithm. You will implement a simplified version of a CLI route planner, that gives you the shortest distance between two nodes, based on existing roadways. This project uses real-world data to do so, but you are welcome to design your own test data first.

1.1 Learning Goals

By the end of this project you will have a better understanding of priority queues, and Dijkstra's famous algorithm. You will also work directly with real world data and develop a sense for how the abstract algorithms we see in class all culminate into this natural application: Priority queues use heaps and hash tables under the hood, which in turn use either trees or linked lists; using priority queues, we can implement a (arguably) pretty cool application on graphs, using Dijkstra's algorithm.

1.2 Use of Other Work

While you may consult any code you have personally written for this class, and any code in the course textbook, you are not allowed to use any other sources, which includes and is not limited to, your peers outside your group, other textbooks, the internet, and AI tools (such as Gemini and ChatGPT).

1.3 Group Work

You may complete this project individually, or you may form a group of up to three people. **I will not be assigning partners for this project.**

If you are in a group, to ensure that everyone is contributing meaningfully to this project, you should each keep track of the contributions of each member's contributions. You will need to describe your contributions (see Section 2).

I expect everyone to contribute meaningfully to **both** the coding and non-coding parts of the project.

Tip 1 *If a member of your group is not working as fast as you would like, the right thing to do would be to help them complete their task (e.g., by explaining things that are not making sense to them) without doing* *the task for them. That may take longer, but doing their part of the project for them would be a disservice to the both of you.*

1.4 Submission

There are two submissions to be made with this project. Both are due by Wednesday, December 18, 11:59PM.

1.4.1 The First Submission

You should submit a ZIP file containing all your code, and unzipping your ZIP file should yield a directory named as follows: If students K. Addams, L. Bigley, and M. Carlson are in the group, the directory should be `addamsk_bigleyl_carlsonm`. In other words, for each member, use the last name following by the initial of their first name, and separate those names using underscores. If you're doing the project by yourself, no need for the underscore.

You will submit

1. `mytests.cpp` – a driver for your test cases.
2. `Makefile` – the makefile.
3. Any other file you create needed for this project.

In your report, your report, include instructions for compiling and running your code.

Each group will make *one* submission (for the code) via Canvas, i.e., only one person in each group will make a submission. This item is called "Final Project - Code Submission" in Canvas. If you are working individually, obviously submit your own code.

1.4.2 The Second Submission

Each student must submit a file called `lastname_firstname_README.pdf` (obviously replacing `lastname` and `firstname` with your own last and first names, respectively). Each student must make their own submission. And each student must write their report independently, i.e., you cannot collaborate on your reports (see more in Section 2). This item is called "Final Project - Individual Reports" in Canvas. More details on this document are given in Section 2.

1.5 Submission Structure

Include a driver file called `mytests.cpp` that runs your various tests. Your tests do not need to test your CLI, but rather the correctness of your priority queue, your graph class, and your implementation of Dijkstra's algorithm. Each test should be contained within a single function, and each such function should be thoroughly commented, describing the purpose of the test and what it demonstrates.

After running your tests, your driver should output how many tests passed and how many tests failed as the last line of its output.

You must include a Makefile that compiles your executables in the following way:

1. `make mytests` should compile and run your tests that demonstrate the correctness of your code.
2. Include instructions in your README on how to execute compile and run your project, using the Makefile.

Your tests should be comprehensive and extensive, while being of reasonable size. Nonetheless, I will also have my own tests that your code will be tested against.

Make sure your code compiles and runs on the department's Linux machines. If your code fails to run or to compile, you will automatically receive a zero for this component of the project.

1.6 Documentation

You must document your code properly. This includes:

1. A full comment header at the top of each source file containing the file name, the developer names, the creation date, and also a brief description of the file's contents.
2. Each method should have a full comment block including a description of what the method does, a "Parameter" section that lists explicit pre-conditions on the inputs to the function, and a "Return" section that explicitly describes the return value and/or any side effects.
3. Make careful use of inline comments to explain various parts of the code that may not be obvious from the code syntax.

2 The README

Each student must submit a README file, which is an essential component of this project, and it is yet another way in which you communicate your technical contributions as a group. A README is a common file included in programs and "code repositories" that explain to a user or to another developer what the directory contains, the purpose of each file in the directory, how the program works, how to run the program, etc. An individual who has never seen your project's description should be able to understand what the project is about and how your solution works solely by reading the README and your comments.

Your README should be organized in the following (brief) sections:

- Project Overview - Give a brief description of your project and the list of contributors.
- Project Structure - Give a description of your submission directory, i.e., the list of files/directories in your project and a brief description of each.
- **[New]** Running the code - Describe how your CLI program should be run when using the terminal.

- Major Design Decisions - Discuss any major design decision made by your group (if any); e.g., if you chose to implement additional functions, or use functional abstractions not described in the project, then you should explain what you did and why you did it.
- Group Challenges - Discuss any major problem your group encountered while implementing the project. If you have no group, skip this section.
- Individual Reflection - Describe your individual contributions to the project and briefly describe any challenges you (not your group) faced in this project.
- Known Issues - If there are any known issues with your code, mention those here. For each issue, try to best explain what may be causing the issue and what would be a potential solution; by thinking through the cause of the issue, you may find a way to fix it :)
- Additional Information - Any additional information you find relevant; keep it brief.

While you cannot collaborate to *write* your READMEs, you are certainly free to discuss elements of the README with your group (if you have one), the TA, and me. However, you cannot produce any written material or recordings during those discussions.

Your submission should be a single PDF file.

Be careful: When writing your README, you may not use text from the project description or other sources verbatim. You can only use your own words. If there is evidence that you received unauthorized help in your reports, you will receive a zero for that part of the assignment and a formal academic integrity violation report will be filed.

3 Design Requirements

In this project, you will only be defining a weighted graph class, which can be based on your prior implementation from Project 4. You will also be implementing a priority queue class that implements the priority queue used in Dijkstra's.

You may also use your own, personally written notes from class or the textbook to guide you through the implementations, and you cannot use any other resources.

Unlike previous projects, this one is more loosely structured. You may structure your classes as you wish. It is up to you to decide if/how your classes should be templated. Nonetheless, the following requirements hold:

1. Make sure you are appropriately handling allocation and deallocation.
2. For each class, you should implement constructors (default and copy), assignment operators, and destructors.
3. Your graph should be a directed weighted graph.

4. Your priority queue should be a min-priority queue that does insert/delete/extract-min in $O(\log n)$ when it has n elements.

5. Your implementation of Dijkstra's algorithm should run in $O((n + m) \log n)$.

I recommend getting familiar with the `std::unordered_map` class from the C++ STL as it will help make many of your implementations a lot easier (it is the equivalent of Python's dictionary class).

3.1 The CLI Program

You may format your CLI program how you wish. It must however obey certain conditions. Upon running your program, you must prompt the user for a file to load. Then, you program should continuously prompt the user for a start vertex s and an end vertex t, both as coordinates. After receiving those inputs, your program should print the coordinates along the shortest s-t path, and it must also print the weight of that path. If the inputs are invalid, you should prompt the user for new coordinates. You should clearly indicate a way to exit the program (e.g., by inputting `q`).

An example run of the program could look as follows (the coordinates below are dummy coordinates):

```
Welcome to my CLI route planner! You may enter
the letter q at any time to exit.

Enter a file name to load: denison.out

Enter a start coordinate: -84.03 43.21
Enter an end coordinate: -84.03 44.21
The shortest path from (-84.03, 43.21) to (-84.03, 44.21) is
<print coordinates here>
and it has weight 1000.

Enter a start coordinate: -84.03 10000
Invalid input!
Enter a start coordinate: q

Exiting... Thank you for using me instead of Google Maps!
```

3.2 Your Data

You are provided with a file in the following format:

```
n m
id_1 x_1 y_1
...
id_n x_n y_n
```

```
u_1 v_1 w_1 s_1
...
u_m v_m w_m s_m
```

In other words, the first line is a pair of natural numbers n and m, where n is the number of vertices and m is the number of edges. The next n lines contain the details about the n nodes, each as three numbers: the first number is a natural number, which is the ID of the current node, the second number is the x-coordinate of the node, and the third number is the y-coordinate of the node. The coordinates are doubles.

Then each of the following m lines is a pair of natural numbers u_i and v_i (both are IDs of nodes) representing edge (u_i, v_i) to be added to the graph followed by a double w_i, which is the weight of the graph to be added, following by a string s_i, which is the name of the edge (i.e., the name of the roadway represented by the edge). You can assume all your weights are positive.

If you wish, you can use the street names in your outputs (instead of printing just the coordinates). If you do that, be careful about how you're parsing things in as those s_i values may contain characters you were not expecting.

You will need to use that file to build your graphs for your CLI application. Of course, if your tests use files of your own design, include those in your submission.

3.2.1 Caveats About the Data

The data is taken from the real world and underwent minimal processing. It is possible that some edges are malformed. If that is the case, simply skip those edges.

3.3 Further Considerations – Completing the Puzzle

You've come a long way from Project 0! Each step along the way has culminated into this project. But, this is not it.

Some simple and natural improvement to the project include (1) specifying which direction to turn to on a given street, (2) including names for the vertices, and naturally (3) using a plotting library to plot the vertices, draw the edges, and highlight the shortest path your algorithm computes, giving a GUI. The level of skill needed to do those tasks however is that of a CS 1 student; this project has you implement the core technical aspects of such an application, so be proud once you complete it!