CS 271 (Spring 2025) Final Project – "You Know You Only Have 30 Minutes": Computing Shortest Paths

Instructor: Prof. Michael Chavrimootoo
Due: Monday, May 12, 2025, 9:00AM – No Extensions

1 Project Overview

In this project, you will be implementing a weighted directed graph class 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 *using real-world data*. Because this project uses real-world data, you are welcome to design your own test data first. This project includes opportunity for extra credit.

1.1 Learning Goals

By the end of this project you will have a better understanding of weighted graphs and Dijkstra's famous algorithm.

You will learn to design your own class and make your own design decisions, building on top of your work throughout the semester.

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 beyond the ones listed in the next paragraph, which includes and is not limited to, your peers outside your group, other textbooks, the internet, and AI tools (such as Gemini and ChatGPT).

External sources you are allowed to use for this project, as with all projects, are `https://stackoverflow.com`, `https://en.cppreference.com`, and `https://cplusplus.com`.

Note: The above list is not exhaustive, and if you use any additional external source, *you must cite it in your report and make a clear reference in your code of where it is used.*

Learning to read documentation, e.g., C++'s, is a *crucial* skill for any CS major, and if you're not comfortable with it, now is a great time to develop that skill.

1.3 Group Work

You can complete this project individually or in pairs. I will leave the pairing if you wish to do it that way. However, if you work in pairs, you must detail your individual contributions in your report (see Section 2.3). I expect everyone to contribute meaningfully to **both** the coding and non-coding parts of the project.

2 Submissions

Projects will be submitted via **Canvas**. Each student must submit, in a zip file, their code base and their report.

2.1 Compatibility

I will compile your code using GCC and C++17 on the Olin machines, so make sure your code compiles and runs correctly on the Olin machines. If your project fails to compile/run, you will receive a zero.

2.2 Code Submission

You are free to design your own classes and data structures. For any class you implement, provide a header file (`.hpp` file) and an implementation file (`.tpp` or `.cpp` file).

You must include a file called `studentTests.cpp` that contains your test cases, which are used to demonstrate that your code works as expected. Your test file should report how many tests you passed and how many you failed. *You will be graded on the quality of your own tests.* Here is how I will be compiling your tests, if all your files are in the same directory

```
g++ -std=c++17 studentTests.cpp -o studentTests
./studentTests
```

Tip 1 *Learning to use* `git` *correctly for both collaborative and personal projects is a tremendous skill to have; you will certainly use it throughout your careers. Remember to make meaningful commit messages throughout the development of the project.*

Moreover, you must document your code properly. This includes:

- 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.
- Each method should have a full comment block including a description of what the method does, a "Parameters" 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.
- Make careful use of inline comments to explain various parts of the code that may not be obvious from the code syntax.

You will be graded on the quality of your code.

2.3 Report Submission

Along with your code, you must submit a report in your zip file as a single PDF file. The report is an essential component of this project, and it is yet another way in which you communicate your technical contributions as a group. 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 report and your comments.

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

- Contributors - List the project contributors, and their individual contributions (if working in pairs).
- Design Decisions - Explain all your design decisions in this project.
- Group Challenges (if applicable) - Discuss any major problem your group encountered while implementing the project. This can also include challenges you encountered with your group.
- Individual Reflection - Describe your individual contributions to the project and briefly describe any challenges you (not your group) faced in this project.
- **[NEW]** Running the Code - Provide detailed instructions on how to compile and run your code in the terminal. If you do not provide those instructions, or if your instructions are incorrect, you will receive a zero.
- 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.
- Further Considerations - See Section 3.3.

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

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

Be careful: When writing your reports, 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 7.

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. The priority queue you use 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` and/or `std::map` classes 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). You are welcome to use other data structures from the C++ STL, or to just use your own implementations from prior projects!

3.1 The CLI Program

You may format your CLI program how you wish. It must however obey certain conditions. These are the conditions your program must obey:

1. Upon running, your program must prompt the user for a file to load.
2. If the file name entered is invalid or the file could not be found, your program should display an appropriate error message and reprompt the user for an input.
3. Your 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
   (a) the coordinates along the shortest s-t path,
   (b) the edge name (see next subsection), and
   (c) the weight of that path.
4. If the coordinate inputs are invalid, your program should reprompt the user for new coordinates.
5. Your program should clearly indicate a way to exit at any time, 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: badfile.out
The file name you've entered could not be found. Try again!

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 and edge names 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!
```

Of course, you are free to customize your program to your taste, as long as you abide by the above requirements. If you do not abide by the above requirements, you will receive a zero.

3.2 Your Data

You are provided with a file called `denison.out` 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.

Be careful when parsing the street names as those s_i values may contain characters you were not expecting. If an edge does not have a name, simply indicate that in your output. If an edge has multiple names, you may print just one of them, or print several of them, indicating why you're printing more than one name.

You will need to use `denison.out` 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 lines/data points are malformed. If that is the case, simply skip those.

3.3 Further Considerations – Completing the Puzzle and Extra Credit

You've come a long way from Project 1! 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, and naturally (2) using a plotting library to plot the vertices, draw the edges, and highlight the shortest path your algorithm computes, giving a GUI application. 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!

For up to an extra 10%, implement meaningful improvements to the project, e.g., the ones above. Your extra credit will depend on the complexity and quality of the improvement.