CS 271 Project 7 – Big Red Graphs

Instructor: Flannery Currin

Due: December 10th, 2025 by 9:00 AM

1 Learning Goals

Graphs are one of the most crucial data structures used in computer science. Many problems can be modeled using graphs (e.g., navigating through a game's grid). In this project, you will continue working with graph representations and use them with real-world data. Specifically, after working on this project you should:

- Be comfortable iteratively extending a partially complete Graph class
- Find the single-source shortest path to all reachable vertices using Dijkstra's algorithm
- Use the results of Dijkstra's algorithm to display the shortest path from a source to a specific destination
- Represent real-world data with a graph

2 Project Overview

You should complete this project in a group – you may select group members or be assigned a group member if you have not picked a group by the class period after this project is released. You are individually responsible for the learning goals above (dividing and conquering may not be the most effective strategy here). Use the Canvas guide on using Github to help manage version control. You may discuss this assignment with your partner(s), the course TA or instructor, but the work you submit must be your group members' own work. You may use your previous projects in this class as a reference.

2.1 Using Coordinate Data and Labeled Edges

I have provided a file called `denison.txt` containing real-world coordinate data from Granville (thank Dr. Chavrimootoo for collecting the data!).

The file follows a format similar to the format we used for Project 6. The first line contains `nv` and `ne` separated by a space. The next `nv` lines contain integer IDs (keys) and floating point x and y coordinates (data to be stored as a tuple). The next `ne` lines represent edges (with a source key, destination key, and weight). The weights are floating point values, so you will need to update the graph class to accepted floats as weights. *Feel free to round these values to 4-5 decimal places when you read them in from the file.*

The main difference with this type of input file is that edges may also have a string label indicating the street (or other landmark). You will need to update the graph class to include a string inside each edge tuple. Use an empty string for unlabeled edges.

Update your previous implementation of `shortestPath()` the appropriate *edge label* (if one exists) and *coordinates* (data) of the vertex at the end of each edge on the path. The first thing printed should be the total length of the path, followed by the coordinates of the source vertex. For example, `shortestPath(73712, 635949)` might print:

```
Total distance: 10
(-82.5183, 40.069)
North Prospect Street(-82.5183, 40.0691)
East College Street(-82.5201, 40.0692)
President Drive(-82.5226, 40.0705)
President's Drive(-82.5227, 40.0708)
President's Drive(-82.523, 40.0715)
Ridge Road(-82.5237, 40.0718)
Ridge Road(-82.5251, 40.0724)
Washington Drive(-82.5251, 40.0725)
Ebaugh Drive(-82.5252, 40.0726)
Ebaugh Drive(-82.5253, 40.0726)
```

How can you find this info given a vertex and its predecessor? Change the Graph class if needed. (**N.B.** it may help to add a key attribute to VertexInfo – iterating through vertices actually only gives us a temporary reference to its values, which is problematic when we have many pointers to predecessors).

2.2 Weighted Shortest Path

Implement Dijkstra's algorithm with a method called `dijkstra(K s)`.

Update your previous implementation of `shortestPath()` to take an additional parameter `weighted`. When `weighted` is `true`, shortestPath() should use `dijkstra` instead of BFS. For example, `shortestPath(73712, 635949, true)` might print:

```
Total distance: 814.397
(-82.5183, 40.069)
North Prospect Street(-82.5183, 40.0691)
East College Street(-82.5201, 40.0692)
President Drive(-82.5226, 40.0705)
President's Drive(-82.5227, 40.0708)
```

```
President's Drive(-82.523, 40.0715)
Ridge Road(-82.5237, 40.0718)
Ridge Road(-82.5251, 40.0724)
Washington Drive(-82.5251, 40.0725)
Ebaugh Drive(-82.5252, 40.0726)
Ebaugh Drive(-82.5253, 40.0726)
```

3 Testing

While it's cool to use real-world data, this is a pretty big data set. To test your implementation, you will want to create smaller, carefully curated test sets and minimal examples. Feel free to use `denison.txt` as a starting point, but create and submit test files with small samples of data intended to test for specific edge or corner cases. Note these in the comments in your updated `graph_tests.cpp` file.

4 Project Submission

On Canvas, before the deadline, you will submit **a zip** containing:

- Your `graph.cpp` file with your C++ implementation.
- `graph.h`
- `graph_tests.cpp` containing your unit tests
- test files you created to represent smaller graphs as you tested your project

5 Grading Scheme

The out-of-class submission will be graded using the following scheme:

Criteria	**Description**	**Points**
Completeness	Meets submission requirements (files, documentation, etc.)	2
Correctness	Passes Dr. Currin's extended tests and follows specs	4
Testing	Includes thorough test cases and thoughtful test files representing subgraphs of the full graph	3
Efficiency	Dijkstra implementation matches expected time complexity	1
Total		**10**

The in-class component is worth 20 points. You should not need to study extra or memorize your out-of-class work for the in-class component. Working on the out-of-class component with the learning goals in mind is your best preparation. The in-class project component lets you test how prepared you are to apply the concepts covered by the project to a new scenario.