CS 271 Project 6 – Graph Search

Instructor: Flannery Currin

Due: November 21st, 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 begin working with graph representations and implementing graph search algorithms. Specifically, after working on this project you should:

- Be comfortable working with both adjacency list and adjacency matrix representations of graphs and be able to convert between them
- Traverse a graph using breadth-first search
- Understand how breadth-first search can be used to find the path between any two vertices containing the fewest edges
- Begin laying the groundwork for shortest path and minimum spanning tree algorithms (spoiler alert for project 7)

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 Representing a Graph

I have provided the beginning of an implementation of a templated `Graph<K, D>` class in `graph.h` and `graph.cpp` along with a starter test file `test_graph_example.cpp` and a file `example.txt` representing a weighted, directed graph. We will not

worry much about edge weights in this project, but we will care much more about them in the next project. The first line of the example input file contains the number of vertices (`nv`) in the graph followed by the number of edges (`ne`). The next `nv` lines of the file contain each vertex's key and then data. The final `ne` lines in the file contain the keys of the vertices the edge is from and to and the `weight` of the edge.

In `graph.h`, a `Struct` is defined to represent information about each vertex, including its **adjacency list**. The `Graph` class has a private member `vertices` which uses `std::map` to associate keys with other vertex information.

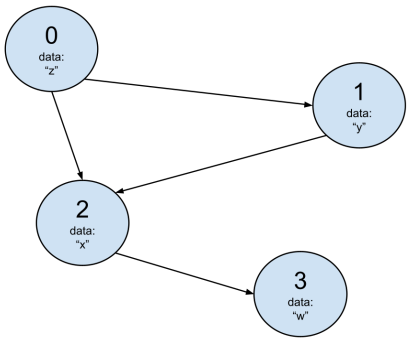
In `graph.cpp`, implement:

- `asAdjMatrix()`: construct an adjacency matrix representation of the graph (a 2D array of ints). The vertex with the smallest key is represented by row/column 0, etc. Use `INT_MAX` to indicate no edge and the weight of an edge otherwise.

2.2 BFS and Shortest Path

`graph.cpp` contains an implementation of depth-first search and an application of it (topological sort as described in CLRS 20.4) along with implementations for several helper methods. Your task is to complete the implementations in `graph.cpp` for:

- `BFS(source)`: fills in the appropriate `VertexInfo` attributes after conducting a Breadth-First Search.
- `shortestPath(s, d)`: calls `BFS(s)` and returns a string representing the shortest path from the vertex with key `s` to the vertex with key `d` (the path from the source to the destination with the fewest edges). For example, for the following graph:



`shortestPath(0, 3)` should return `0->2->3`

If there is no path from the specified source to the specified destination, return the empty string.

You may define additional helper methods if you would like, but do not change the provided methods and do not remove anything from the header file.

3 Testing

Test your implementations thoroughly. Construct different graphs using a similar representation to the file provided or using some other clear representation. Try graphs with different properties. For example, you can create an undirected graph from a directed graph by inserting an edge from vertex v2 to v1 any time you insert an edge from v1 to v2.

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 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 graphs	3
Efficiency	BFS/shortest path 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.