Code Project 2 Design Document

Anthony Streich, Keegan Erickson, Dawson Wright

November 1, 2024

1 Overall Summary

This document details the code of a directed graph that utilizes a priority queue and an adjacency list in order to perform breadth-first search from a text file given to the program. Also included are headers for our queues and stacks that our program utilizes.

2 digraph Implementation

The `DiGraph` class implements a directed graph using an adjacency list representation. It allows for the addition of edges, retrieval of adjacent vertices, and implementation of Dijkstra's algorithm for finding the shortest path from a source vertex to a destination vertex.

Upon creation of a graph object a value for graph size is taken in as an argument for creation of the adjacency list two-dimensional vector. The vector is a reference list for all integer vertex numbers that themselves contain a list of each edge structure that contains the pointed to vertex with its weight.

Addition of edges is done by a void function that takes in the source and pointed to vertices with edge weight. It also checks if the vertex is out of range for the graph.

3 Path Finding

Path finding has two methods available; a dijkstra's algorithm for searching graphs with weighted edges and a breadth first search (BFS) for unweighted edged graphs.

The BFS class object was designed to take in graph object as implemented in the Digraph class of the digraph.h file. The BFS object also takes in a source vertex on creation. The BFS algorithm initials at instanciation of the BFS class and can call upon functions that check if a vertex is connected to the sources, as well as a function that will provide a shortest path to the target vertex.

in the Digraph class itself is a Dijkstra algorithm void function for finding the shortest path from the source to destination vertices. A priority queue is

used to select the next vertex with the smallest weights that updates a parent array that tracks the previous vertex. The method prints to terminal the total weight of the path and displays the path taken. A stack utilizing a self resizing array was used to trace the path backwards and flip it to start from source and destination in the output to terminal.

The priority queue was integral to the function of the Dijksta algorithm. The queue was constructed using a resizing array, expanding as the need arises, that was used to store heap nodes. The heap nodes held a Node structure that held each item paired with its priority. Which, in this implementation means that the priority held the edge weight. Lower weighted edges held a higher priority in the queue. Heap up and heap down functions were used to restructure specific nodes up or down after an item was removed. Peek functions were used to get the values and priorities before removal with a removeMin() function.

4 main.cpp

This document describes the implementation of all above outlined graph objects and methods used to find the shortest path from one node to another were implemented in main.cpp file. The main function was designed to read graph data from a text file, provide a complete graph object when given the text data, and include a terminal interface that took in user input to specify what source and destination vertices that the user wanted to use to find a shortest path, if available. The program also logs the events and execution time. Specifically logged was the total program run time as well as length of execution time of the Dijkstra path finding.

5 Project Problems Encountered

The main problem we encountered during the creation of this project is that we started with two different implementations for our code, one using an adjacency list and one with an adjacency matrix. This used up a lot of our time for the project because we were trying to figure out which implementation would be the best for our situation. Eventually, we decided to go with the adjacency list because the matrix implementation utilized for too much space and time for what was needed for the project and it had problems with trying to find the shortest path through breadth-first search. It was not until almost the end that it was realized that the BFS could not find the shortest path in a graph. With little time, a Dijkstra path finding algorithm was implemented right before the due date for this project.

What was decided as a strategy going forward in future group coding projects was to more carefully divide up tasks that can be done separately. More work was done than was necessary.