Dijkstra's Algorithm Implementation Report

COP 4530 Programming Project 4

Rishabh Bhargav, Ishfat Abrar Islam, Yamin Arafat Islam, Daniel Misherky

1. Introduction

This project implements an undirected weighted Graph ADT and performs Dijkstra's algorithm to find the shortest path between two vertices. The implementation includes a custom priority queue data structure and provides methods for adding and removing vertices and edges, as well as calculating shortest paths.

2. Design Decisions

2.1 Graph Representation

An adjacency list representation was chosen for the graph implementation. Each vertex maintains a list of edges connecting to other vertices. This approach offers:

- Space Efficiency: O(V + E) space complexity, where V is vertices and E is edges

- Easy Edge Traversal: Simple iteration through neighbors during Dijkstra's algorithm

2.2 Data Structures

- Vertex Structure: Contains a label and a vector of Edge objects

- Edge Structure: Stores destination vertex label and edge weight

- Graph Storage: Uses a 'std::map<std::string, Vertex>' for O(log n) vertex lookup

2.3 Priority Queue Implementation

A custom min-heap priority queue was implemented using a vector-based binary heap. This ensures:

- O(log n) insertion and extraction operations

- O(1) access to the minimum element

- Efficient distance updates during Dijkstra's algorithm

The priority queue nodes store:

- Vertex label

- Current shortest distance from source

- Previous vertex in the shortest path

3. Algorithm Implementation

3.1 Dijkstra's Algorithm Overview

Dijkstra's algorithm finds the shortest path from a source vertex to all other vertices in a weighted graph with non-negative edges.

Key Steps:

1. Initialize distances: source = 0, all others = infinity

2. Add all vertices to priority queue with their initial distances

3. While priority queue is not empty:

   - Extract vertex with minimum distance

   - For each neighbor, calculate new distance = current distance + edge weight

   - If new distance < current distance, update it

4. Reconstruct path by following previous pointers from destination back to source

3.2 Implementation Details

The 'shortestPath()' method implements Dijkstra's algorithm:

unsigned long shortestPath(string startLabel, string endLabel, vector<string> &path)

Time Complexity: O((V + E) log V) where V is vertices and E is edges

Space Complexity: O(V) for distance maps, previous map, and priority queue

3.3 Path Reconstruction

After finding shortest distances, the path is reconstructed by:

1. Starting at the destination vertex

2. Following the previous map backwards

3. Building the path vector in reverse order

4. Reversing to get the correct order: source → destination

4. Testing and Results

4.1 Test Case

The implementation was tested using the provided example:

Vertices: { "1", "2", "3", "4", "5", "6" }

Edges:

- ("1", "2", 7), ("1", "3", 9), ("1", "6", 14)

- ("2", "3", 10), ("2", "4", 15)

- ("3", "4", 11), ("3", "6", 2)

- ("4", "5", 6)

- ("5", "6", 9)

4.2 Results

Query: Shortest path from vertex "1" to vertex "5"

Output:

- Shortest Distance: 20

- Path: 1 → 3 → 6 → 5

Verification:

- Path 1→3→6→5: 9 + 2 + 9 = 20

- This is the minimum distance

5. Edge Cases Handled

The implementation properly handles various edge cases:

- Duplicate Vertices: Prevents adding vertices with duplicate labels

- Self-Loops: Prevents edges from a vertex to itself

- Duplicate Edges: Checks for existing edges before adding

- Non-Existent Vertices: Validates vertex existence before edge operations

- Same Start and End: Returns distance 0 and path containing only that vertex

6. Conclusion

This project successfully implements a Graph ADT with Dijkstra's algorithm for finding shortest paths. The adjacency list representation provides efficient graph operations, and the custom min-heap priority queue ensures optimal performance. All required methods are implemented, tested, and produce correct results.