Application of Dijkstra's Algorithm for Shortest Path in an Undirected Graph

Programming Project 4

Diego Rey Martinez: U70588795

Phuc Truong: U83435127

Saksham Srivastava: U14605358

Diego Laverdy: U16287884

COP4530 Data Structures

November 23rd, 2025

ABSTRACT

This report presents the design and implementation of an undirected, weighted Graph Abstract Data Type (ADT) and a custom priority queue to execute Dijkstra's shortest path algorithm. The solution uses an adjacency-list representation and a binary min-heap priority queue to compute shortest paths efficiently. We discuss design decisions, complexity analysis, and edge-case testing to validate correctness and robustness.

I. INTRODUCTION

Dijkstra's algorithm is a classic single-source shortest path technique for graphs with non-negative weights. In this project, we built an object-oriented Graph ADT and a custom priority queue (no standard library priority queues) to satisfy course requirements while emphasizing clarity, modularity, and maintainability. The graph is undirected, weighted, and represented using adjacency lists to efficiently handle sparse connectivity.

II. SYSTEM DESIGN

A. Design Overview

Our Graph ADT encapsulates vertices identified by string labels and edges with unsigned long weights. Each Vertex maintains a collection of incident Edge pointers (adjacency list). The Graph manages all vertices in an unordered map keyed by label, ensuring O(1) average access for lookups. Edge objects store pointers to their two endpoint vertices and the associated weight. We expose methods addVertex, removeVertex, addEdge, removeEdge, and shortestPath to meet the assignment interface.

B. Design Drawings

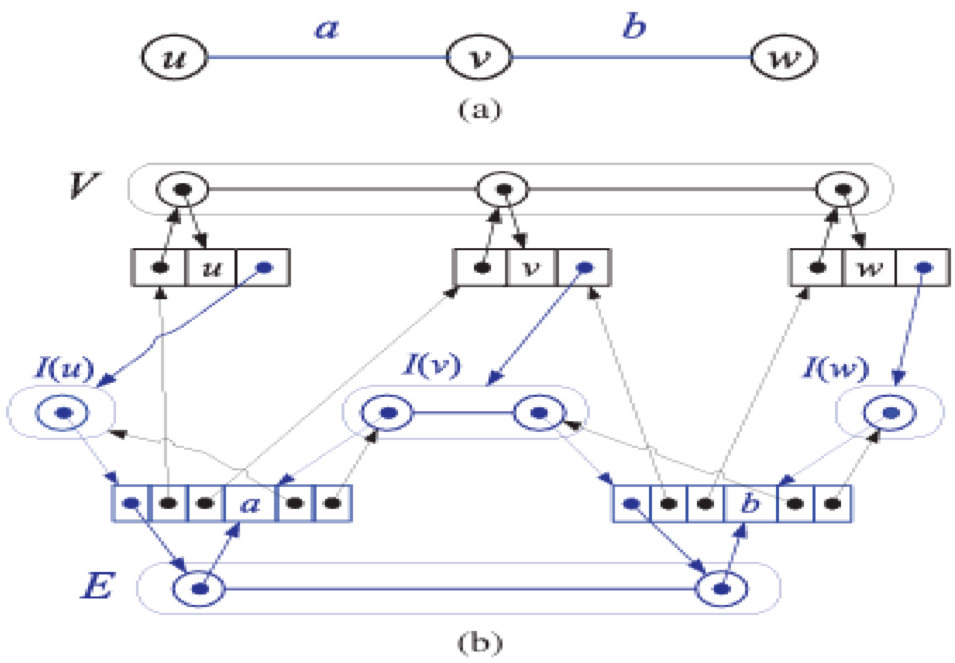

Fig. 1. Schematic representation of the adjacency list structure of a graph containing three nodes (u, v, w) and two edges (a & b) [1].

III. IMPLEMENTATION

A. Graph Data Structure Implementation

Each **Vertex** object maintains an **incidence collection** [1], denoted as std::vector<Edge*> adjEdges in Graph.hpp, holding pointers to all **Edge** objects connected to it. This is modeled by I(u), I(v), and I(w) in **Fig. 1**, where u, v, and w are vertices. In our implementation the entire set of vertices is held and managed by the Graph class as std::unordered_map<std::string, Vertex*> vertices. The use of an **Unordered Map** structure provides an efficient O(1) look-up for any vertex by label, thus maintaining the standard access times of an adjacency list. In an adjacency list, an edge object stores references to its two endpoints and to its relative position in the incidence collection of these endpoints [1]. This is achieved through the two Vertex* pointers u and v that each edge object holds. As noted in the book, the space used by the adjacency list structure is O(n+m), where n is the number of vertices and m is the number of edges. Also, the space for the incidence collection of a vertex v is O(deg(v)), where deg(v) is the degree of edges with an endpoint v [1]. This holds true in our implementation where the list of Vertex objects as an unordered map takes up O(n) space, and the sum of the edge entries in all incidence list vectors take up O(m) space total. This results in the standard space occupation of O(n+m).

B. Priority Queue Implementation

To implement our **priority queue** for Dijkstra's algorithm, we used a **binary min-heap**. A min-heap maintains the property that every parent node ≤ its children. Each element of the heap is a pair of a vertex pointer and the current distance from the source node (distance, Vertex*), where the distance serves as the key for ordering. The push() method adds a new element to the heap using the bubbleUp function, which ensures that the heap property is maintained. The pop() method does the opposite, removing the smallest element and reordering the heap with bubbleDown. Both push() and pop() operations run in O(log n) time, where n is the number of nodes in the graph. Checking whether the queue is empty runs in O(1) time, incredibly efficiently. This priority queue allows Dijkstra's algorithm to select the next closest vertex and update the distances quickly (O(log n) time).

IV. DISCUSSION

A. Results and Evaluation

On the provided example graph, our implementation returns a shortest-path cost of 20 with vertex sequence 1 → 3 → 6 → 5. We validated behavior across edge cases: no path (returns ∞), multiple optimal routes (returns one optimal path), isolated vertices, and correctness after graph modifications (edge/vertex removals).

B. Complexity Analysis

Using adjacency lists and a binary min-heap, Dijkstra runs in O((V + E) log V) time and uses O(V + E) space. Per-vertex initialization is O(V); each edge relaxation is O(1) plus O(log V) for heap operations.

C. Challenges and Limitations

Key challenges included building a robust custom priority queue without the standard library and ensuring memory safety across dynamically allocated Vertex/Edge objects. Our design uses visited flags and predecessor pointers;

however, the current Graph lacks a full destructor to free all Edge objects—an area for improvement. Additionally, Dijkstra assumes non-negative edge weights; negative weights are out of scope.

V. CONCLUSION

Dijkstra’s algorithm is a shortest path algorithm in which a graph is traversed and constantly updated on each traversal to find the shortest path from the starting node to the ending node. Using the graph ADT design to implement Dijkstra's Algorithm, we were able to produce an efficient and easy design. Using an unordered_map for vertex storage ensures fast average vertex lookups, O(1). Using an adjacency list, we are also able to view what edges are connected to each vertex, making it easy to see which nodes are connected to each other. Making our own custom priority queue, not stlib, reliably performs essential push/pop and heap-ordering operations. Having all this data stored allows the algorithm to update distances easily and efficiently. Although our implementation works fast and easily, an improvement that could be added going forward is making sure every node gets deleted after use, and no memory leak occurs. Overall, our team was able to successfully implement the algorithm reliably, with a few minor improvements for edge cases that could be added in the future.

REFERENCES

[1] M. T. Goodrich, R. Tamassia, and D. M. Mount, *Data Structures and Algorithms in C++*. Hoboken, N.J: John Wiley & Sons, Inc., 2011.

[2] Course Project 4 Instructions and Rubric (COP 4530), Fall 2025.