Dijkstra’s Algorithm: Final Project Report

Aubrey Kopkau, Chloe Robinson, Laray Lopez, Mark Anjoul

University of South Florida

COP4530

23 November 2025

Introduction

Dijkstra's algorithm is a graph search technique used to determine the shortest path between nodes in a non-negative weighted graph. It follows the greedy method, meaning that at each stage, the algorithm chooses the best immediate option; in this case, it is the vertex with the smallest known distance to the source. By repeatedly selecting the nearest unvisited vertex, Dijkstra's algorithm constructs the minimum spanning tree for all reachable nodes. This algorithm is widely utilized in real-life applications, such as traffic information systems and routing software. This report explores the implementation of Dijkstra's algorithm using an adjacency list, maintaining an $O(|V| + |E|)\log|V|)$ runtime and supporting the algorithm's need for quick access to each vertex's immediate neighbors.

Correctness of Dijkstra's Algorithm

Two lemmas prove the correctness of Dijkstra's algorithm. They focus on the upper bound and extracting the vertex that maintains the true shortest distance.

Lemma 1 is the upper bound property, or the algorithm invariant. It states that for all vertices, the current distance estimate is greater than or equal to the true shortest path distance from the source vertex, d[v] $\geq \delta(s, v)$. Dijkstra's algorithm never underestimates the true shortest path, the estimated distance starts off at ∞, which is too large, and decreases through relaxation, but it is not artificially small.

Lemma 2 refers to the extraction of the vertex that has the minimum, though perhaps temporary, distance. It states $d[u] = \delta$(s,u) , or relaxation cannot reduce the distance estimate of vertex that is already settled. Once a vertex is extracted, the shortest-path distance does not change again, upholding the greedy algorithm correctness. A future path cannot skip ahead to a shorter path, therefore the estimate $d[u]$ must already be equal to the true shortest distance.

Dijkstra's Algorithm Implementation

Graph Design

The Graph ADT was built around an adjacency list data structure. In this structure, each vertex stores a list of adjacent vertices, as well as their weights. Adjacency lists provide us with a space complexity of $O(|V| + |E|)$ for the algorithm. This representation not only reduces the space used by this algorithm but also optimizes its time complexity. With such a structure, all vertices of a graph can be traversed in $O(|V| + |E|)$ time using a breadth-first search (BFS) method. The graph utilizes methods such as addVertex(), removeVertex(), addEdge(), and removeEdge(), that do just as their names indicate. We implemented a shortestPath() function, which effectively uses Dijkstra's algorithm to locate the shortest path to a given vertex.

Organization

When implementing Dijkstra’s algorithm, classes provide a way to model a graph with its components. This then allows the algorithm to be cleaner, modular, and easy to maintain. The classes encapsulate information about nodes, connections, and edge weights. This aids the graph in being able to represent itself through adjacency lists stored within the class. From this organization, the algorithm can efficiently access each vertex’s neighbors and edge weights when checking for a new shortest path. Through grouping data, classes assist Dijkstra's algorithm by enabling it to perform on a well-defined graph structure.

We have split the project into six major files:

- Graph.hpp, which defines the graph class, its data structures, and methods.
- Graph.cpp, which implements the graph functionality.
- GraphBase.hpp, which provides the abstract base class specifying the required graph interface and virtual functions.
- PriorityQueue.hpp, which declares the priority queue class and its supporting structures.
- PriorityQueue.cpp, which implements the priority queue operations.
- Main.cpp, which handles user interaction for the algorithm and serves as a testing space to ensure functionality.

Priority Queue in Dijkstra’s Algorithm

After traversing the graph with BFS, a min heap is used to store the vertices for which the shortest distance is not finalized yet. The min heap is implemented as a priority queue to get the minimum distance vertex from a set of not-yet-included vertices.

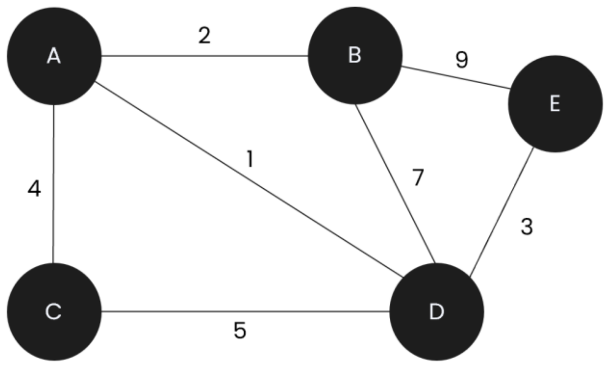
The priority queue is crucial to improving the efficiency of Dijkstra’s algorithm. It keeps track of the next vertex to be processed. Instead of scanning all vertices to find the minimum distance, which would result in O(V^2), the priority queue takes the vertex with the lowest cost. Each time a shorter path is found, the distance is updated and pushed back into the queue. This ensures the algorithm processes the vertices in the most optimal order to keep the overall runtime low.

In the main loop, Dijkstra's algorithm continuously takes the vertex with the smallest distance from the source and checks adjacent vertices to lower the path costs. When a vertex is removed from the priority queue and processed, the shortest path is finalized. Then it continues to check each neighboring vertex, ensuring that the current path is the shortest. If a shorter path is found, the distance is updated, and the vertex is added back into the priority queue. This continues until all vertices have been processed, and the queue is empty.

Example Functionality

See the figure below:

Figure 1: Weighted Graph Example



Assuming we begin at A, our shortestPath function first initializes A to zero, and all other values to infinity. The priority queue now has one value of {A(0)}.

Each main loop iteration goes as follows:

1. **Searching A**: The algorithm identifies A's neighboring vertices and inserts them into the priority queue. It updates the distances to D(1), B(2), and C(4), adding them to the priority queue in that order. A is now visited.
2. **Searching D**: Next in the queue, the algorithm identifies D's unvisited neighboring vertices and inserts them into the priority queue. It updates the distance to E(4), adding it to the queue. The priority queue now contains B(2), C(4), and E(4).
3. **Searching B:** Next in the queue, the algorithm identifies that B has no unvisited neighboring vertices, and no shorter path. The priority queue now contains C(4) and E(4).
4. **Searching C:** Next in the queue, the algorithm identifies that C has no unvisited neighboring vertices, and no shorter path. The priority queue now contains E(4).
5. **Searching E:** Next in the queue, the algorithm identifies E's unvisited neighboring vertices and inserts them into the priority queue. Once again, there are no updates. This was the last node in the priority queue, so the loop terminates.

The algorithm was designed to trace back to find the shortest path, then reverse it to get the normal start-to-end structure.

Challenges

One of the primary challenges we encountered involved deciding on the most appropriate graph representation for implementing the algorithm. We debated on using an adjacency matrix structure rather than the list we decided on, specifically because of the constant time edgelookups; however, that came at the cost of significantly more memory with an O(V^2) space complexity. In a sparse graph, this would waste a significant amount of space over the iterations. Balancing these pros and cons were an important part of the final design our implementation.

Conclusion

Dijkstra's algorithm and it's supporting lemmas ensure the correctness of the shortestPath() function, demonstrated through our adjacency list and binary heap priority queue implementation. Its average time complexity of $O((|V| + |E|) \log |V|)$ confirms its efficiency in handling large graphs, while the adjacency list structure provides a space complexity of $\mathrm{O}(|\mathrm{V}| + |\mathrm{E}|)$, highlighting its efficient use of memory.

References

Goodrich, M., Tamassia, R., & Mount, D. (2011). *Data Structures and algorithms in C++, second edition*. John Wiley and Sons.