# Dijkstra's Algorithm

Authors:

Dean Worrels

Jason Maya

Yahya Khlifa

Sujal Kumar Shah

April 20th 2025

## Dijkstra's Algorithm

In this project, we implemented Dijkstra's Algorithm to solve the classic single-source path problem on an undirected, weighted graph. The goal is to find the most efficient path from a starting node to a target node, minimizing the total path weight. Dijkstra's algorithm is applied in digital mapping services, social networking applications, as well as telephone networks.

Dijkstra's algorithm is a well-known example of the greedy design pattern. It works by expanding the shortest known paths from a source vertex outward. At every step, it chooses the nearest unvisited vertex and updates the shortest paths to its neighbors using a method called edge relaxation. The algorithm uses a priority selection approach to efficiently determine which vertex to visit next, ensuring we always explore the optimal path first. Each vertex is assigned a tentative distance value. The starting vertex is assigned zero, and infinity for the other vertices. As the algorithm proceeds, these values are updated to reflect the shortest path found so far. Once a vertex is marked as visited, its distance label is finalized because no shorter path will be found later. This behavior makes Dijkstra's algorithm particularly powerful in graphs with non-negative weights, as it guarantees that each shortest path is found without revisiting nodes.

### Graph Representation

An adjacency list is a graph representation technique where each vertex keeps a list of its neighbors and the edge weights connecting them. Instead of using a full matrix to represent all possible connections, this method only stores existing edges, making it efficient, especially for

sparse graphs. In our project, each vertex is stored as a vertex struct that includes a string label and a vector of edge structs. Each edge holds the label of the neighbor and the weight of its connection. To locate vertices by label, we implemented a helper function called findVertexIndex(). Instead of using a hash map, we can scan the vertex list linearly to get the index associated with a given label. It's not the most optimal method, but it satisfies the constraint of only using the class <string> and class <vector> from standard libraries.

**HeapQueue Implementation**

To maintain the efficiency of the greedy selection strategy in Dijkstra's algorithm, we implemented a custom HeapQueue class based on a complete binary tree using a vector. The internal class VectorCompleteTree helps maintain the binary heap structure, offering operations such as addLast, removeLast, swap, and access to parent and child nodes. The HeapQueue class provides methods for inserting elements, accessing the minimum, and removing the minimum. We defined a comparator template to maintain head order based on distance, allowing the minimum distance node to always be on top. This structure mimics the functionality of priority_queue without relying on STL.

**Algorithm Logic**

The core logic of Dijkstra's algorithm is implemented in the shortestPath function. We initialize all vertex distances to a large constant and set the source vertex distance to zero. A visited array is used to keep track of which vertices have already been finalized. During each iteration, the algorithm removes the vertex with the smallest tentative distance from the HeapQueue. Once selected, it relaxes the edges to its neighbors by checking if the current path to

each neighbor is shorter than any previously found path. If a shorter path is discovered, we update the neighbor's distance and store the current node as its predecessor, and reinsert the neighbor into the heap with the updated distance. After all reachable vertices have been visited, we use the prev vector to backtrack from the destination to the start, reconstructing the shortest path. The function returns both the path and the total distance.

**Results**

We ran our program in the terminal to test the functionality and correctness of our graph implementation and shortest path algorithm. We tested our implementation on two distinct graphs to evaluate correctness and performance. The first graph used numeric labels for vertices "1" to "6" and contained a variety of weighted edges. When we ran Dijkstra's algorithm on this graph to find the shortest path from vertex "1" to vertex "5", the algorithm correctly identified the path with a total distance of 20. This result was consistent with our manual calculations and confirmed that the shortest path logic and edge relaxation process were working as intended. The second graph was designed to resemble a simplified campus map, with vertex labels representing buildings such as ENB, LIB, CAS, and SUN. We used this more contextual example to verify that the algorithm could handle real-world naming conventions and distances. In one test case, the shortest path from ENB to SUN was computed, totaling 2885 in distance. Another test from LIB to CAS yielded the path with a distance of 1532. These results demonstrated the algorithm's ability to navigate a realistic network and confirm shortest-path accuracy under varying topologies.

## Conclusion

Through this project, we demonstrated how Dijkstra's algorithm can be effectively implemented from scratch using only a limited set of C++ standard library classes. BY developing a custom min-heap structure to replace STL's priority queue, we maintained efficient path selection while understanding how priority-based algorithms operate internally. The project highlights the practical relevance of Dijkstra's greedy approach in real-world scenarios such as routing and network analysis. While the current implementation is limited to undirected graphs with non-negative weights, it provides a solid foundation for future improvements. Expanding the algorithm to handle directed or weighted graphs with negative edges, optimizing heap operations, or introducing more flexible graph structures would enhance both its functionality and scalability. These additions would allow for broader applications and more robust performance in complex graph environments.

**Referenecs**

Goodrich, Michael T., and Roberto Tamassia. *Data Structures and Algorithms in C++*. 2nd ed., Wiley, 2011.

GeeksforGeeks. (2022, September 23). *Applications of Dijkstra's shortest path algorithm*. GeeksforGeeks. https://www.geeksforgeeks.org/applications-of-dijkstras-shortest-path-algorithm/