Network Router Simulator

Min-Heap Dijkstra with Custom Graph ADT

Krishna Chemudupati

1 Overview

This project is a network router simulation built in C++ that computes shortest paths using Dijkstra's algorithm. All core data structures, including a min-heap priority queue, an adjacency list graph, and a stack, are implemented from scratch without relying on STL containers.

2 Architecture

The project is organized into modular compilation units. The foundational types are defined in `data_structures.h`, which contains the core struct definitions for `VERTEX`, `NODE`, `HEAP`, and `STACK`. The min-heap priority queue is implemented in `heap.cpp` and `heap.h`, providing the priority queue that drives Dijkstra's algorithm. Path reconstruction is handled by the stack module in `stack.cpp` and `stack.h`, which reverses the predecessor chain into the correct source-to-target order. The graph module in `graph.cpp` and `graph.h` contains the Dijkstra implementation along with adjacency list operations such as printing, path output, and distance queries. Finally, `main.cpp` and `main.h` handle command-line argument parsing, graph construction from the input file, and the interactive command loop.

3 Algorithm

3.1 Dijkstra's Shortest Path

The implementation follows the textbook Dijkstra's algorithm with a min-heap priority queue, achieving $O((V+E)\log V)$ time complexity. The algorithm begins by initializing all vertex distances to ∞ and the source distance to 0, with every vertex colored WHITE to indicate it has not yet been discovered.

The main loop repeatedly extracts the minimum-key vertex from the heap and examines each of its neighbors. When a neighbor is WHITE, meaning it has never been visited, the algorithm relaxes the edge, marks the neighbor GRAY, and inserts it into the heap. When a neighbor is already GRAY, meaning it is currently in the heap, the algorithm relaxes the edge and performs a decrease-key operation to update its position in the heap if a shorter path was found. Neighbors that are BLACK have already been finalized and are skipped entirely.

The algorithm terminates when the heap is empty in the single-source case, or as soon as the target vertex is extracted from the heap in the single-pair case. The three-color scheme (WHITE, GRAY, BLACK) tracks vertex states throughout this process to avoid redundant processing.

3.2 Edge Relaxation

The relaxation step checks whether the path through vertex u offers a shorter route to vertex v than the currently known distance. If so, it updates the distance and predecessor, and triggers a decrease-key operation if the vertex is already in the heap.

```c
void relax(pVERTEX u, pVERTEX v, double w, pHEAP heap) {
    if (v->key > u->key + w) {
        v->key = u->key + w;
        v->pi = u->index;
        if (v->color == GRAY && v->position > 0)
            decreaseKey(heap, v->position, v->key);
    }
}
```

4 Data Structures

4.1 Min-Heap Priority Queue

The priority queue is implemented as an array-based binary min-heap. It supports insertion with a bubble-up procedure, extraction of the minimum element followed by a heapify to restore the heap property, and an in-place decrease-key operation that updates a vertex's key and bubbles it upward. All three operations run in $O(\log V)$ time. Each vertex stores its current position within the heap array, which enables $O(1)$ lookup when a decrease-key is needed. This position-tracking mechanism is critical for achieving the overall $O((V + E)\log V)$ bound on Dijkstra's algorithm.

4.2 Adjacency List Graph

The graph is represented as an array of linked lists, `ADJ[1..n]`, where each entry stores the outgoing edges from that vertex. Each edge is stored as a `NODE` struct containing the source vertex u, destination vertex v, weight w, and an edge index. The adjacency list supports configurable insertion order—edges can be inserted at the front or rear of each list depending on a command-line flag, which affects the traversal order during Dijkstra's execution.

4.3 Stack

The stack is an array-based implementation used exclusively for path reconstruction. After Dijkstra completes, the predecessor chain is followed from the target vertex back to the source, pushing each vertex onto the stack along the way. The stack is then popped to print the vertices in the correct source-to-target order.

5 Features

5.1 Graph Topologies

The simulator supports both directed and undirected graph topologies, selectable via a command-line argument. When an undirected graph is specified, each edge (u, v, w) from the input file is stored as two directed edges: one from u to v and one from v to u, both with weight w.

5.2 Routing Modes

Two routing modes are available. The single-source mode computes the shortest paths from a given source vertex to every other vertex in the graph by running Dijkstra's algorithm until the heap is fully exhausted. The single-pair mode computes only the shortest path between two specific vertices and terminates early as soon as the target vertex is extracted from the heap, avoiding unnecessary computation.

5.3 Query Commands

The program operates through an interactive command loop that reads instructions from standard input. The `PrintADJ` command displays the full adjacency list representation of the graph. The `SingleSource` command takes a source vertex *s* and computes shortest paths to all reachable vertices. The `SinglePair` command takes a source *s* and target *t* and computes only the shortest path between them. After a computation has been performed, `PrintPath` outputs the sequence of vertices along the shortest path from *s* to *t*, and `PrintLength` outputs the total distance of that path. The `Stop` command terminates the program.

6 Usage

The project is built using the provided Makefile and invoked with three command-line arguments. The first argument is the path to the network topology input file. The second argument specifies the graph type as either `DirectedGraph` or `UndirectedGraph`. The third argument is a flag set to `0` for front insertion or `1` for rear insertion when constructing the adjacency lists.

```
make
./PJ3 <InputFile> <GraphType> <Flag>
```

6.1 Input File Format

The input file begins with two integers specifying the number of vertices and the number of edges. Each subsequent line defines an edge with its index, source vertex, destination vertex, and weight.

```
<num_vertices> <num_edges>
<edge_index> <u> <v> <weight>
...
```

6.2 Example

```
$ ./PJ3 network01.txt DirectedGraph 1
SingleSource 1
PrintPath 1 8
The shortest path from 1 to 8 is:
[1:    0.00]-->[4:    5.00]-->[5:    7.00]-->[3:   11.00]-->[6:
   18.00]-->[7:   23.00]-->[8:   26.00].
PrintLength 1 8
The length of the shortest path from 1 to 8 is:     26.00
Stop
```

7 Complexity Analysis

The overall time complexity of Dijkstra's algorithm with the min-heap is $O((V + E)\log V)$. This arises from each of the V vertices being inserted into and extracted from the heap at $O(\log V)$ cost, combined with up to E edge relaxations that may each trigger a decrease-key operation, also at $O(\log V)$ cost. Graph construction runs in $O(V + E)$ time, and path reconstruction requires $O(V)$ time in the worst case.

In terms of space, the adjacency list requires $O(V + E)$ storage for the vertex array and all edge nodes. The min-heap and stack each require $O(V)$ space, as they hold at most one entry per vertex. The total space complexity is therefore $O(V + E)$.

8 Build

The project is compiled using `g++` with the `-Wall` flag for strict compiler warnings. It has no external dependencies beyond the C++ standard library. Running `make` builds the executable, and `make clean` removes all compiled object files and the executable.

```
make        # Build the project
make clean  # Remove compiled objects and executable
```