Dijkstra's algorithm

Dijkstra's algorithm finds the **shortest path distances** from a single source node to every other node in a graph with **non-negative edge weights**. It returns the minimum cost to reach each node (and can be adapted to recover actual shortest paths).

How it works

Start at the source with distance 0. Repeatedly pick the not-yet-finalized node with the smallest tentative distance, finalize it, and relax its outgoing edges (use that node to try to improve neighbors' tentative distances). Continue until every reachable node is finalized or the target is finalized.

Core idea

Maintain a set of nodes whose shortest distance from the source is known (finalized). Use a priority queue (min-heap) keyed by tentative distances to always extract the closest unsettled node next. Relax edges from that node to update neighbors' tentative distances.

Step-by-step process

1. Initialize a distance array dist[] with ∞ for all nodes except dist[source] = 0.
2. Put (0, source) into a min-heap (priority queue).
3. While the heap is not empty:
    - Extract (d, u) — the node u with smallest tentative distance d.
    - If d > dist[u] skip (this is an outdated entry).
    - For each edge (u -> v) with weight w:
        - If dist[u] + w < dist[v] then set dist[v] = dist[u] + w and push (dist[v], v) into the heap.
4. After the loop, dist[v] holds the shortest distance from source to v (or ∞ if unreachable).

pseudocode

```
function Dijkstra(source, adjacency):
    for each node v:
        dist[v] = INF
        parent[v] = -1
    dist[source] = 0
    minheap = priority queue of pairs (dist, node)
    push (0, source) into minheap
```

```
while minheap not empty:
    (d, u) = pop minheap
    if d > dist[u]:
        continue
    for (v, w) in adjacency[u]:
        if dist[u] + w < dist[v]:
            dist[v] = dist[u] + w
            parent[v] = u
            push (dist[v], v) into minheap

return dist, parent;
```

Implementation Link:

algorithm-/Dijkstra at main · Jannat651/algorithm-

Problem 1: Dijkstra

Problem Link: Problem - C - Codeforces

Problem Statement

You are given a **weighted undirected graph** with n vertices and m edges.
Each edge connects two vertices and has a positive cost.

Objective: Determine **one valid shortest path** from vertex 1 to vertex n.

- If no such path exists, output -1.
- If a path exists, output **all vertices on the shortest path in order**.

Hint

- All edge weights are positive → **Dijkstra's algorithm** is appropriate for computing minimum distances.
- To recover the **actual path**, use a **predecessor array** to trace the route backward from vertex n.

Solution Approach

1. Construction of the Graph

- Use an **adjacency list** for efficient storage (O(n + m)) for up to 10^5 vertices and edges.
- For each input edge (a, b, w):
  - Add (b, w) to the neighbors of a.
  - Add (a, w) to the neighbors of b.
- This preserves the **undirected structure**.

2. Initialization

Three main components are required:

1. **Distance array dist[]**
   - Stores the shortest known distance to each vertex.
   - Initialize all entries to a **very large number**.
   - Set dist[1] = 0 for the starting vertex.
2. **Predecessor array prev[]**
   - Stores the **previous vertex** on the best path discovered so far.
   - Initialize all entries to -1.
3. **Priority queue (min-heap)**
   - Stores pairs (distance, vertex).
   - Always extracts the vertex with the **smallest tentative distance**.

3. Dijkstra's Algorithm Logic

1. While the priority queue is not empty:
   - Extract (u, dist[u]) with the **minimal tentative distance**.
   - For each neighbor (v, w) of u:
     - Compute candidate_distance = dist[u] + w.
     - If candidate_distance < dist[v]:
       - Update dist[v] = candidate_distance.
       - Update prev[v] = u.
       - Insert (dist[v], v) into the priority queue.
   - Ignore **outdated queue entries** where dist[u] is already smaller.

4. Verification of Reachability

- After processing all reachable vertices:
  - If dist[n] remains **infinite**, vertex n cannot be reached → output -1.
  - Otherwise, a **shortest route** exists and can be reconstructed.

5. Path Reconstruction Steps

1. Begin at vertex n.
2. Follow the prev[] array backward:
3. n → prev[n] → prev[prev[n]] → ... → 1
4. Reverse the collected sequence to get the **correct order** from vertex 1 to vertex n.
5. Output all vertices in the path.

6. Complexity and Suitability

- **Time Complexity:** O((n + m) log n) using a min-heap → efficient for the given constraints.
- **Memory Usage:** Efficient due to adjacency lists and simple auxiliary arrays.

Pseudocode

```
function Dijkstra(n, adjacency, source=1, target=n):
    for each vertex v in 1..n:
        dist[v] = INF
        prev[v] = -1

    dist[source] = 0
    minheap = priority queue of (dist, vertex)
    push (0, source) into minheap

    while minheap is not empty:
        (d, u) = pop minheap
        if d > dist[u]:
            continue   // outdated entry

        for each (v, w) in adjacency[u]:
            candidate = dist[u] + w
            if candidate < dist[v]:
                dist[v] = candidate
                prev[v] = u
                push (dist[v], v) into minheap

    if dist[target] == INF:
        print -1
    else:
        path = empty list
        current = target
        while current != -1:
            path.append(current)
            current = prev[current]
        reverse(path)
        print path
```

Implementation Link:

algorithm-/Dijkstra/Dijkstra?/dijkstra_1.cpp at main · Jannat651/algorithm-