Dijkstra?

Dijkstra's algorithm is a classic method for finding the single-source shortest path in weighted graphs with non-negative edge weights. It works by exploring vertices in order of their currently known shortest distance from the source, gradually expanding the search frontier until all reachable vertices are processed or the target vertex is reached. In this implementation, the algorithm starts from vertex 1 and aims to reach vertex $n$along the shortest path. Key data structures used include an adjacency list for efficient graph representation, a min-heap (priority queue) to select the closest unvisited vertex, distance and parent arrays to track optimal paths, and a visited set to avoid redundant processing. The adjacency list is especially important for handling large graphs (up to 100,000 vertices and edges), as it provides fast access to neighbors while using memory proportional to the number of edges. In each iteration, the algorithm extracts the vertex with the smallest distance from the priority queue and updates the distances of its neighbors if a shorter path is found. An important optimization is early termination when the target vertex $n$is reached, reducing unnecessary computations. The parent array allows reconstruction of the shortest path by backtracking from vertex $n$to vertex 1. For example, in a graph with vertices 1 through 5 and six edges, the algorithm correctly identifies the shortest path 1→4→3→5 with total weight 5, even though alternative paths like 1→2→5 or 1→2→3→5 have higher weights (7). The implementation also handles multiple edges between vertices and self-loops by always choosing the minimum weight during relaxation. If no path exists from vertex 1 to n, the distance to $n$remains infinite, resulting in an output of -1, as required.

Steps for Dijkstra's Algorithm:

Phase 1: Setup and Initialization

First, represent the graph with vertices and their weighted connections, so each vertex knows its neighbors and the cost to reach them. Prepare the necessary tracking structures: a distance array initialized to infinity for all vertices except the start vertex, a predecessor array to remember the path, a visited set to mark processed vertices, and a priority queue that always returns the closest unvisited vertex.

Phase 2: Algorithm Execution

Start from the source vertex (vertex 1) by setting its distance to 0 and adding it to the priority queue. Then, repeatedly process vertices as follows: select the vertex with the smallest known distance from the queue, mark it as visited to avoid reprocessing, and check if it is the target

vertex $n$for possible early termination. For each neighbor of the current vertex, calculate the total distance through the current vertex. If this new distance is shorter than the previously recorded distance, update the neighbor's distance, set the current vertex as its predecessor, and add the neighbor to the priority queue with the updated distance.

Phase 3: Result Determination
After completing the algorithm, check whether a path exists by examining the distance of the target vertex. If it is still infinity, no path exists; otherwise, a shortest path exists. To reconstruct the path, start from the target vertex and repeatedly follow the predecessor pointers backward, collecting all vertices along the way. Finally, reverse the collected sequence to obtain the shortest path from the start vertex to the target vertex.

Input:

5 6

1 2 2

2 5 5

2 3 4

1 4 1

4 3 3

3 5 1

Execution Steps:

1. **Read Input:** n = 5, m = 6. Create an empty adjacency list for vertices 1 to 5.

2. **Build Graph:** Add edges bidirectionally with weights: 1–2 (2), 2–5 (5), 2–3 (4), 1–4 (1), 4–3 (3), 3–5 (1).

3. **Initialize Data:** Distance array: [0, ∞, ∞, ∞, ∞], Parent array: [-1, -1, -1, -1, -1], Visited array: [false, false, false, false, false], Priority queue: vertex 1 with distance 0.

4. **Process Vertex 1:** Remove vertex 1 (distance 0), mark visited. Update neighbors: vertex 2 distance = 2 (parent 1), vertex 4 distance = 1 (parent 1). Queue: vertex 4 (1), vertex 2 (2).

5. **Process Vertex 4:** Remove vertex 4 (1), mark visited. Neighbor 1 already visited. Update vertex 3 distance = 4 (parent 4). Queue: vertex 2 (2), vertex 3 (4).

6. **Process Vertex 2:** Remove vertex 2 (2), mark visited. Neighbor 1 visited. Update vertex 5 distance = 7 (parent 2). Vertex 3 new distance = 6, no update (current distance 4). Queue: vertex 3 (4), vertex 5 (7).

7. **Process Vertex 3:** Remove vertex 3 (4), mark visited. Neighbors 2 and 4 visited. Update vertex 5 distance = 5 (better than 7), set parent = 3. Queue: vertex 5 (5), vertex 5 (7).

8. **Process Vertex 5:** Remove vertex 5 (5), mark visited. Target reached, stop algorithm.

9. **Check Result:** Distance to vertex 5 = 5 → path exists.

10. **Reconstruct Path:** Backtrack from vertex 5 → parent 3 → parent 4 → parent 1. Path backwards: 5, 3, 4, 1. Reverse to get shortest path: 1, 4, 3, 5.

11. **Output Result:** 1 4 3 5.

Output:

1 4 3 5

Pseudocode:

READ n, m

CREATE graph[n+1] as adjacency list

```
FOR each edge:
    READ a, b, w
    ADD (b, w) to graph[a]
    ADD (a, w) to graph[b]

dist = [INF] * (n+1)
parent = [-1] * (n+1)
visited = [false] * (n+1)
pq = min-heap

dist[1] = 0
PUSH (0, 1) to pq

WHILE pq is not empty:
    (d, u) = POP from pq
    IF visited[u]:
        CONTINUE
    visited[u] = true
    IF u == n:
        BREAK

    FOR each (v, w) in graph[u]:
        IF NOT visited[v] AND d + w < dist[v]:
            dist[v] = d + w
            parent[v] = u
            PUSH (dist[v], v) to pq
```

```
IF dist[n] == INF:
    PRINT -1
ELSE:
    path = []
    v = n
    WHILE v != -1:
        ADD v to path
        v = parent[v]
    REVERSE path
    PRINT path
```

Here is the solution in code for Dijkstra:

https://github.com/Hazra32/Algorithm-Problem/blob/main/Dijkstra/Dijkstra%3F/dijkstra.cpp