Dijkstra?

Dijkstra's Algorithm – Shortest Path (Vertex 1 to n)

Dijkstra's algorithm is a fundamental graph traversal technique designed to solve the **single-source shortest path problem** in weighted graphs with **non-negative edge weights**. The algorithm systematically explores vertices in order of their **current best-known distance** from the source vertex, gradually expanding the search frontier until all reachable vertices are processed or the target vertex is found.

In this implementation, the algorithm starts at **vertex 1** and aims to reach **vertex n** through the **shortest path**. Several key data structures are used for efficiency:

- **Adjacency List** – represents the graph efficiently and provides O(1) access to neighbors.
- **Priority Queue (min-heap)** – ensures the closest unvisited vertex is processed next.
- **Distance Array** – tracks the shortest known distance to each vertex.
- **Parent/Predecessor Array** – remembers the previous vertex for path reconstruction.
- **Visited Set** – prevents reprocessing vertices.

The algorithm works in multiple iterations:

1. Extract the vertex with the **smallest distance** from the priority queue.
2. Mark it as **visited**.
3. If it is the **target vertex n**, stop early.
4. Explore all neighbors and **relax edges**:
    - Compute the distance to the neighbor through the current vertex.
    - If this distance is smaller than the known distance:
        - Update the neighbor's distance.
        - Update the neighbor’s predecessor to the current vertex.

        - Add the neighbor to the priority queue.

The **parent array** allows path reconstruction by backtracking from **vertex n** to **vertex 1**.

Example: For vertices 1 through 5 with six edges, the algorithm finds the optimal path `1 → 4 → 3 → 5` with total weight 5, despite alternatives like `1 → 2 → 5` or `1 → 2 → 3 → 5` (weight 7). Multiple edges or self-loops are handled by always choosing the **minimum weight** during relaxation. If no path exists, the distance to **vertex n** remains **INF**, triggering output `-1`.

Steps for Dijkstra

Phase 1: Setup and Initialization

1. **Graph Representation**
    - Represent vertices with their neighbors and edge weights.
2. **Tracking Structures**
    - Distance array (∞ for all except start).
    - Parent/predecessor array.
    - Visited set.
    - Priority queue (min-heap).

Phase 2: Algorithm Execution

3. **Start from Source**
    - `distance[start] = 0`
    - Add start vertex to priority queue.
4. **Main Processing Loop**
    - **Select:** Take vertex with smallest distance.
    - **Mark:** Mark vertex as visited.

- **Check Destination:** Stop if target vertex n reached.
- **Explore Neighbors:**
  - Compute distance to neighbor via current vertex.
  - If smaller than known distance:
    - Update distance.
    - Set current vertex as predecessor.
    - Push neighbor into priority queue.

Phase 3: Result Determination

5. **Path Existence Check**
   - If `distance[n] == INF` → no path exists.
   - Else → shortest path exists.
6. **Path Reconstruction**
   - Start at vertex n.
   - Follow predecessor pointers backward.
   - Collect all vertices along the way.
   - Reverse sequence to get path from start to end.

Input:

5 6

1 2 2

2 5 5

2 3 4

1 4 1

4 3 3

3 5 1

Execution:

Step 1: Read Input

• Read n = 5, m = 6

• Create an empty graph with 6 positions (for vertices 1 to 5)

Step 2: Build Graph

• For edge 1-2 weight 2: connect vertex 1 to 2 with weight 2, and vertex 2 to 1 with weight 2

• For edge 2-5 weight 5: connect vertex 2 to 5 with weight 5, and vertex 5 to 2 with weight 5

• For edge 2-3 weight 4: connect vertex 2 to 3 with weight 4, and vertex 3 to 2 with weight 4

• For edge 1-4 weight 1: connect vertex 1 to 4 with weight 1, and vertex 4 to 1 with weight 1

• For edge 4-3 weight 3: connect vertex 4 to 3 with weight 3, and vertex 3 to 4 with weight 3

• For edge 3-5 weight 1: connect vertex 3 to 5 with weight 1, and vertex 5 to 3 with weight 1

Step 3: Initialize Data

• Distance array: vertex 1=0, vertices 2-5=infinity

• Parent array: all vertices = -1 (no parent)

• Visited array: all vertices = false (not visited)

• Priority queue: add vertex 1 with distance 0

Step 4: Process Vertex 1

• Remove vertex 1 from queue (distance 0)

• Mark vertex 1 as visited

• Check neighbor vertex 2: new distance 0+2=2, update distance to 2, set parent to 1, add to queue

• Check neighbor vertex 4: new distance 0+1=1, update distance to 1, set parent to 1, add to queue

• Queue now has: vertex 4 (distance 1), vertex 2 (distance 2)

Step 5: Process Vertex 4

• Remove vertex 4 from queue (distance 1)

• Mark vertex 4 as visited

• Check neighbor vertex 1: already visited, skip

• Check neighbor vertex 3: new distance 1+3=4, update distance to 4, set parent to 4, add to queue

• Queue now has: vertex 2 (distance 2), vertex 3 (distance 4)

Step 6: Process Vertex 2

• Remove vertex 2 from queue (distance 2)

• Mark vertex 2 as visited

• Check neighbor vertex 1: already visited, skip

• Check neighbor vertex 5: new distance 2+5=7, update distance to 7, set parent to 2, add to queue

• Check neighbor vertex 3: new distance 2+4=6, but current distance is 4, so no

update

• Queue now has: vertex 3 (distance 4), vertex 5 (distance 7)

Step 7: Process Vertex 3

• Remove vertex 3 from queue (distance 4)

• Mark vertex 3 as visited

• Check neighbor vertex 2: already visited, skip

• Check neighbor vertex 4: already visited, skip

• Check neighbor vertex 5: new distance 4+1=5, better than current 7, update

distance to 5, set parent to 3, add to queue

• Queue now has: vertex 5 (distance 5), vertex 5 (distance 7)

Step 8: Process Vertex 5

• Remove vertex 5 from queue (distance 5)

• Mark vertex 5 as visited

• Vertex 5 is the target, so stop the algorithm

Step 9: Check Result

• Distance to vertex 5 is 5 (not infinity), so path exists

Step 10: Reconstruct Path

• Start from vertex 5

• Parent of 5 is 3 • Parent of 3 is 4

• Parent of 4 is 1

• Path backwards: 5, 3, 4, 1

• Reverse path: 1, 4, 3, 5

Step 11: Output Result

• Print: 1 4 3 5

Output:

1 4 3 5

Pseudocode:

```
READ n, m

CREATE graph[n+1]


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 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
```

Code:

https://github.com/shanto470/algorithm_plm/blob/main/Dijkstra/Dijkstra%3F/Dijkstra.cpp