Dijkstra's Shortest Path Algorithm: Step-by-Step Guide with Example

Dijkstra's algorithm is a famous algorithm for finding the shortest paths from a source vertex to all other vertices in a weighted graph with non-negative edge weights.

Algorithm Steps

1. **Initialize Distances and Visited Nodes**:
    - Set the initial distance for the source node as 0 and for all other nodes as infinity (∞).
    - Mark all nodes as unvisited.
2. **Select the Node with Minimum Distance**:
    - From the set of unvisited nodes, select the node with the smallest known distance.
3. **Update Distances**:
    - For the current node, update the distance of each unvisited neighbor by checking if the path from the source to the neighbor through the current node is shorter than the known distance. If so, update it.
4. **Mark the Current Node as Visited**:
    - Mark the current node as visited, so it won't be checked again.
5. **Repeat**:
    - Repeat steps 2–4 until all nodes are visited or no unvisited nodes remain with finite distances.
6. **End**:
    - The algorithm ends when all reachable nodes have been visited. The shortest path distances from the source to each node are then determined.

Example

Let's go through an example with a graph.

Graph Representation (Weighted, Directed):

Suppose we have a graph with the following edges and weights:

From	To	Weight
A	B	4
A	C	1
B	C	2
B	D	5
C	D	8
C	E	10
D	E	2
E	F	3

Let's find the shortest path from node A to all other nodes.

Step-by-Step Execution

1. **Initialization**: Set initial distances from A as follows:

Distances: A=0, B=∞, C=∞, D=∞, E=∞, F=∞

2. **Step 1 - Start at A**: The initial node is A with a distance of 0.
    - Update distances for B and C:
        - A → B: 0 + 4 = 4
        - A → C: 0 + 1 = 1

Distances: A=0, B=4, C=1, D=∞, E=∞, F=∞

Visited: A

3. **Step 2 - Move to C (Minimum Distance)**:
    - Current node C with distance 1.
    - Update distances for B, D, and E:
        - C → B: 1 + 2 = 3 (shorter than 4, so update B)
        - C → D: 1 + 8 = 9
        - C → E: 1 + 10 = 11

Distances: A=0, B=3, C=1, D=9, E=11, F=∞

Visited: A, C

4. **Step 3 - Move to B (Minimum Distance)**:
    - Current node B with distance 3.
    - Update distances for D:
        - B → D: 3 + 5 = 8 (shorter than 9, so update D)

Distances: A=0, B=3, C=1, D=8, E=11, F=∞

Visited: A, C, B

5. **Step 4 - Move to D (Minimum Distance)**:
    - Current node D with distance 8.
    - Update distances for E:
        - D → E: 8 + 2 = 10 (shorter than 11, so update E)

Distances: A=0, B=3, C=1, D=8, E=10, F=∞

Visited: A, C, B, D

6. **Step 5 - Move to E (Minimum Distance)**:

- Current node E with distance 10.
- Update distances for F:
  - E → F: 10 + 3 = 13

Distances: A=0, B=3, C=1, D=8, E=10, F=13

Visited: A, C, B, D, E

7. **Step 6 - Move to F (Minimum Distance)**:
   - Current node F with distance 13.
   - No updates needed as all reachable nodes have been visited.

Distances: A=0, B=3, C=1, D=8, E=10, F=13

Visited: A, C, B, D, E, F

Final Shortest Path Distances from A

- **A → A**: 0
- **A → B**: 3
- **A → C**: 1
- **A → D**: 8
- **A → E**: 10
- **A → F**: 13

These distances represent the shortest paths from node A to all other nodes in the graph.

Java Code:

```
import java.util.*;

class Node implements Comparator<Node> {
   public int node;
   public int cost;

   public Node() {}

   public Node(int node, int cost) {
      this.node = node;
      this.cost = cost;
   }

   @Override //The @Override annotation in Java is used to indicate that a method is overriding a method from a superclass or interface.
   public int compare(Node n1, Node n2) {
```

```
        return Integer.compare(n1.cost, n2.cost);
    }
}

public class Dijkstra {

    public static void dijkstra(int[][] graph, int source) {
        int n = graph.length;
        int[] distances = new int[n];
        boolean[] visited = new boolean[n];

        // Initialize distances
        Arrays.fill(distances, Integer.MAX_VALUE);
        distances[source] = 0;

        // Priority queue to get the node with the minimum distance
        PriorityQueue<Node> queue = new PriorityQueue<>(n, new Node());
        queue.add(new Node(source, 0));

        while (!queue.isEmpty()) {
            int current = queue.poll().node;

            // If already visited, skip it
            if (visited[current]) continue;
            visited[current] = true;

            // Visit all the neighbors of the current node
            for (int i = 0; i < n; i++) {
                if (graph[current][i] != 0 && !visited[i]) { // Check if neighbor exists
                    int newDist = distances[current] + graph[current][i];
                    if (newDist < distances[i]) {
                        distances[i] = newDist;
                        queue.add(new Node(i, newDist));
                    }
                }
            }
        }

        // Print the shortest distances from the source
        System.out.println("Shortest distances from node " + source + ":");
        for (int i = 0; i < n; i++) {
            System.out.println("To node " + i + " is " + distances[i]);
        }
    }

    public static void main(String[] args) {
        // Example graph represented as an adjacency matrix
```

```
        int[][] graph = {
            {0, 4, 1, 0, 0, 0},
            {4, 0, 2, 5, 0, 0},
            {1, 2, 0, 8, 10, 0},
            {0, 5, 8, 0, 2, 0},
            {0, 0, 10, 2, 0, 3},
            {0, 0, 0, 0, 3, 0}
        };

        int source = 0; // Starting node
        dijkstra(graph, source);
    }
}
```

Output:

Shortest distances from **node 0**:

To node 0 is 0

To node 1 is 3

To node 2 is 1

To node 3 is 8

To node 4 is 10

To node 5 is 13