Assignment No:3
Name : Prathmesh Pattewar
Prn:12420193

Title: Dijkstra Algorithm.

Problem Statement: Write a program to find the shortest path using Dijkstra Equation for Link State Routing Protocol which is used by Open Shortest Path First Protocol (OSPF) in the Internet for the network flow provided by instructor.

Course Objective: To learn networking standards, IP packet switching and routing used in the Internet.

Course Outcome: Build wired and wireless intranet with correct communication and service frameworks.

Tools Required: Eclipse, Java.

Theory:

1. Link State Routing Protocol:

Link State Routing Protocols are dynamic routing protocols used in packet-switched networks to determine the best path for data packets. The most widely used link-state protocol is OSPF (Open Shortest Path First), which operates within an Autonomous System (AS). OSPF maintains a link-state database of the entire network topology and computes the shortest path using Dijkstra's Algorithm.

Link state routing is a technique in which each router shares the knowledge of its neighborhood with every other router in the internetwork.

- It is a dynamic routing algorithm in which each router shares knowledge of its neighbors with every other router in the network.
- A router sends its information about its neighbors only to all the routers through flooding.
- Information sharing takes place only whenever there is a change.
- It makes use of **Dijkastra's Algorithm** for making routing tables.
- **Problems –** Heavy traffic due to flooding of packets.
  – Flooding can result in infinite looping which can be solved by using **Time to live (TTL)** field.

2. Dijkstra's Algorithm:

Dijkstra's algorithm is a graph search algorithm that finds the **shortest path** between a source node and all other nodes in a weighted graph. It is optimal and guarantees the least-cost path in a network.

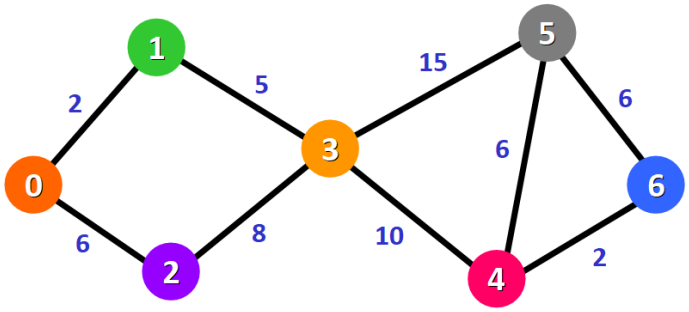
2.1 Working of Dijkstra's Algorithm in OSPF:

- Each router in the network identifies its neighbors and calculates the cost (typically delay, bandwidth, or hop count) to reach them.
- Routers exchange **Link State Advertisements (LSAs)** to share their topology information.
- Each router independently constructs a **Link State Database (LSDB)** representing the entire network graph.
- Dijkstra's algorithm is then applied on the LSDB to calculate the **Shortest Path Tree (SPT)**.
- The result is stored in the routing table, where each destination has the next-hop and associated cost.

2.2 Steps in Dijkstra's Algorithm:

- Initialize distances from source to all nodes as infinity, except the source (distance 0).
- Maintain a priority queue (or visited/unvisited set).
- Select the unvisited node with the smallest known distance.
- Update distances to adjacent nodes if a shorter path is found.
- Mark the current node as visited and repeat until all nodes are visited.

3. Example:



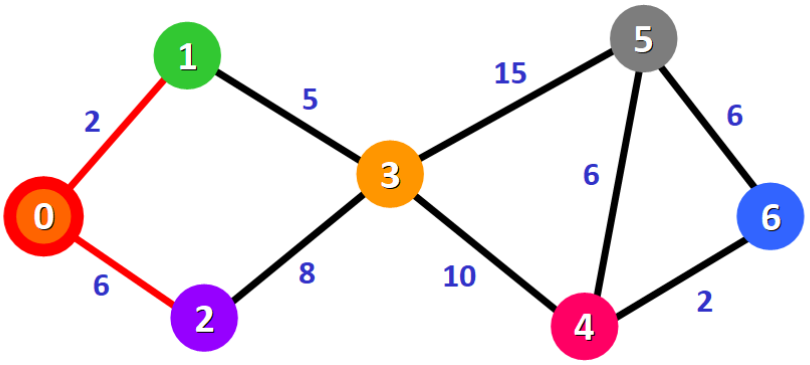
Initially, we have this list of distances (please see the list below):

- The distance from the source node to itself is 0. For this example, the source node will be node 0 but it can be any node that you choose.
- The distance from the source node to all other nodes has not been determined yet, so we use the infinity symbol to represent this initially.

Unvisited Node: {0,1,2,3,4,5,6}

0	1	2	3	4	5	6
0	∞	∞	∞	∞	∞	∞

Since we are choosing to start at node 0, we can mark this node as visited. Equivalently, we cross it off from the list of unvisited nodes and add a red border to the corresponding node in diagram:
Now we need to start checking the distance from node 0 to its adjacent nodes. As you can see, these are nodes 1 and 2 (see the red edges):



Unvisited Node:{1,2,3,4,5,6}

0	1	2	3	4	5	6
0	**2**	**6**	∞	∞	∞	∞

Unvisited Node:{2,3,4,5,6}

0	1	2	3	4	5	6
0	**2**	**6**	**7**	∞	∞	∞

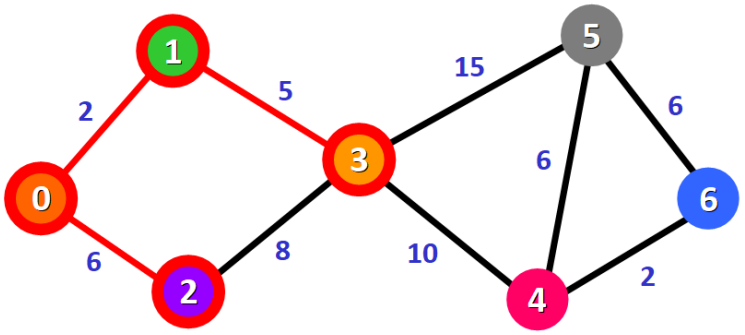
For node 3: the total distance is **7** because we add the weights of the edges that form the path 0 -> 1 -> 3 (2 for the edge 0 -> 1 and 5 for the edge 1 -> 3).
From the list of distances, we can immediately detect that this is node 2 with distance **6**:

Unvisited Node:{3,4,5,6}

0	1	2	3	4	5	6
0	**2**	**6**	**7**	∞	∞	∞

Node 3 already has a distance in the list that was recorded previously (**7,** see the list below). This distance was the result of a previous step, where we added the weights 5 and 2 of the two edges that we needed to cross to follow the path 0 -> 1 -> 3.
But now we have another alternative. If we choose to follow the path 0 -> 2 -> 3, we would need to follow two edges 0 -> 2 and 2 -> 3 with weights **6** and **8**, respectively, which represents a total distance of **14**.
Clearly, the first (existing) distance is shorter (7 vs. 14), so we will choose to keep the original path 0 -> 1 -> 3. **We only update the distance if the new path is shorter.**

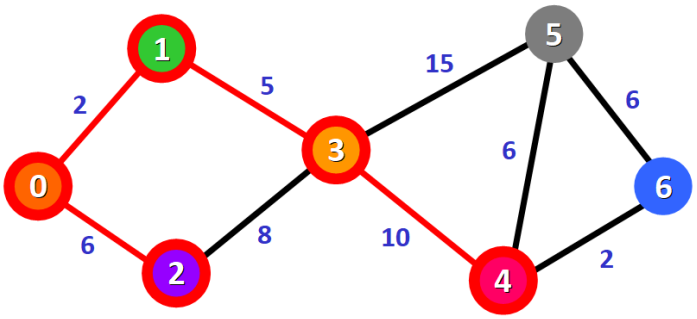
Therefore, we add this node to the path using the first alternative: 0 -> 1 -> 3.



We update the distances of these nodes to the source node, always trying to find a shorter path, if possible:
For node 4: the distance is **17** from the path 0 -> 1 -> 3 -> 4.
For node 5: the distance is **22** from the path 0 -> 1 -> 3 -> 5.
Notice that we can only consider extending the shortest path (marked in red). We cannot consider paths that will take us through edges that have not been added to the shortest path (for example, we cannot form a path that goes through the edge 2 -> 3).
Unvisited Node: {4,5,6}

0	1	2	3	4	5	6
0	2	**6**	**7**	**17**	**22**	∞

We need to choose which unvisited node will be marked as visited now. In this case, it's node 4 because it has the shortest distance in the list of distances. We add it graphically in the diagram:



For node 5:

- The first option is to follow the path 0 -> 1 -> 3 -> 5, which has a distance of **22** from the source node (2 + 5 + 15). This distance was already recorded in the list of distances in a previous step.
- The second option would be to follow the path 0 -> 1 -> 3 -> 4 -> 5, which has a distance of **23** from the source node (2 + 5 + 10 + 6).

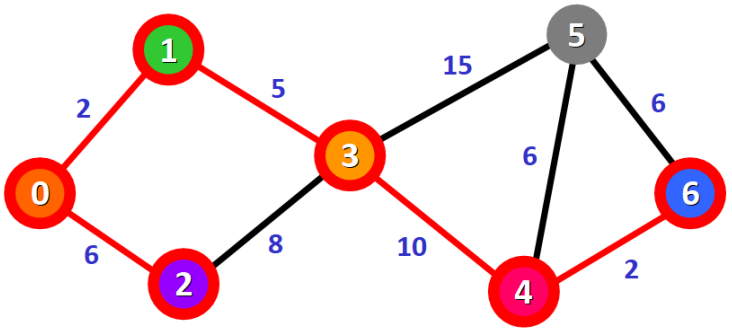
Clearly, the first path is shorter, so we choose it for node 5.

For node 6:

- The path available is 0 -> 1 -> 3 -> 4 -> 6, which has a distance of **19** from the source node (2 + 5 + 10 + 2).

Unvisited Node:{5,6}

0	1	2	3	4	5	6
0	**2**	**6**	**7**	**17**	**22**	**19**



Only one node has not been visited yet, node 5. Let's see how we can include it in the path. There are three different paths that we can take to reach node 5 from the nodes that have been added to the path:

- **Option 1:** 0 -> 1 -> 3 -> 5 with a distance of **22** (2 + 5 + 15).
- **Option 2:** 0 -> 1 -> 3 -> 4 -> 5 with a distance of **23** (2 + 5 + 10 + 6).
- **Option 3:** 0 -> 1 -> 3 -> 4 -> 6 -> 5 with a distance of **25** (2 + 5 + 10 + 2 + 6).

Unvisited Node:{6}

0	1	2	3	4	5	6
0	**2**	**6**	**7**	**17**	**22**	**19**

We have the final result with the shortest path from node 0 to each node in the graph.

Program:

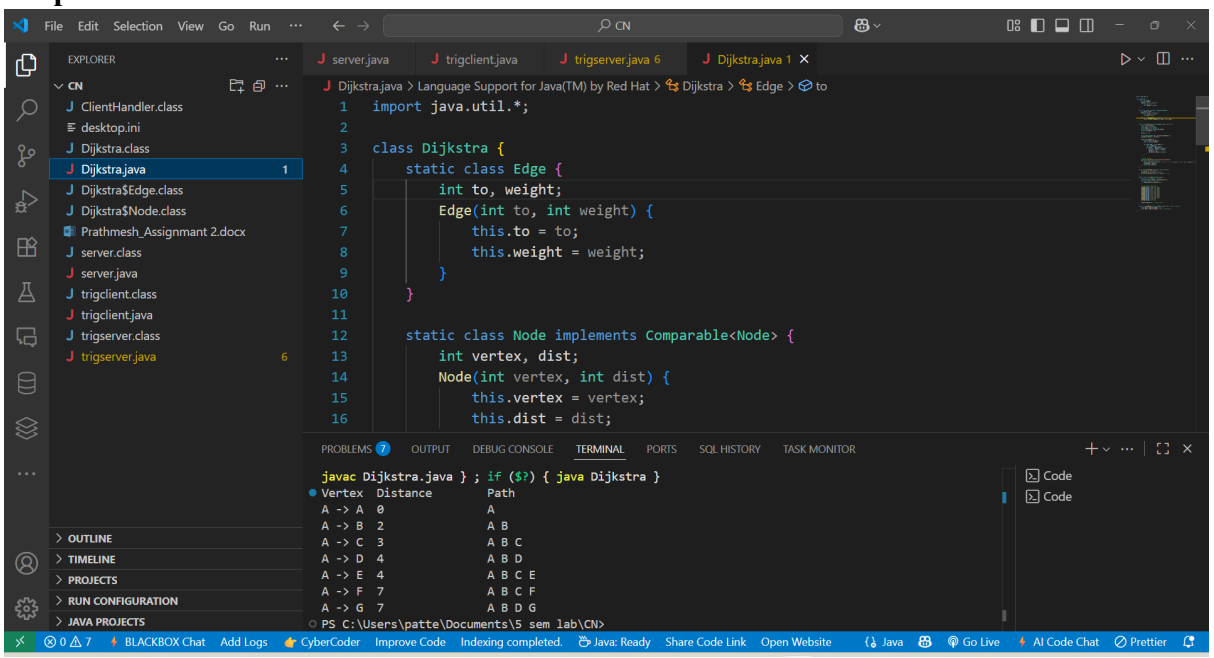
```
import java.util.*;

class Dijkstra {
    static class Edge {
        int to, weight;
        Edge(int to, int weight) {
            this.to = to;
            this.weight = weight;
```

```
        }
    }

    static class Node implements Comparable<Node> {
        int vertex, dist;
        Node(int vertex, int dist) {
            this.vertex = vertex;
            this.dist = dist;
        }
        public int compareTo(Node other) {
            return Integer.compare(this.dist, other.dist);
        }
    }

    static void dijkstra(List<List<Edge>> graph, int src) {
        int V = graph.size();
        int[] dist = new int[V];
        int[] parent = new int[V];
        Arrays.fill(dist, Integer.MAX_VALUE);
        Arrays.fill(parent, -1);

        dist[src] = 0;

        PriorityQueue<Node> pq = new PriorityQueue<>();
        pq.add(new Node(src, 0));

        while (!pq.isEmpty()) {
            Node current = pq.poll();
            int u = current.vertex;

            for (Edge edge : graph.get(u)) {
                int v = edge.to;
                int weight = edge.weight;
                if (dist[u] + weight < dist[v]) {
                    dist[v] = dist[u] + weight;
                    parent[v] = u;
                    pq.add(new Node(v, dist[v]));
```

```
                }
            }
        }

        // Print results
        System.out.println("Vertex\tDistance\tPath");
        for (int i = 0; i < V; i++) {
            System.out.print((char)('A' + src) + " -> " +
(char)('A' + i) + "\t" + dist[i] + "\t\t");
            printPath(i, parent);
            System.out.println();
        }
    }

    static void printPath(int vertex, int[] parent) {
        if (vertex == -1) return;
        printPath(parent[vertex], parent);
        System.out.print((char)('A' + vertex) + " ");
    }

    public static void main(String[] args) {
        int V = 7; // number of vertices
        List<List<Edge>> graph = new ArrayList<>();
        for (int i = 0; i < V; i++) {
            graph.add(new ArrayList<>());
        }

        // Add edges (undirected)
        addEdge(graph, 0, 1, 2);
        addEdge(graph, 0, 2, 5);
        addEdge(graph, 1, 2, 1);
        addEdge(graph, 1, 3, 2);
        addEdge(graph, 2, 3, 3);
        addEdge(graph, 2, 4, 1);
        addEdge(graph, 2, 5, 4);
        addEdge(graph, 3, 4, 2);
        addEdge(graph, 3, 6, 3);
```

```
        addEdge(graph, 4, 6, 5);
        addEdge(graph, 5, 6, 2);

        dijkstra(graph, 0); // Start from A
    }

    static void addEdge(List<List<Edge>> graph, int u, int v,
int w) {
        graph.get(u).add(new Edge(v, w));
        graph.get(v).add(new Edge(u, w)); // undirected
    }
}
```

Output:



Conclusion: In this program, we implemented Dijkstra's Algorithm using an adjacency list and a priority queue (min-heap) to efficiently determine the shortest paths from a given source node to all other nodes in a weighted graph.