**Distance Vector Routing Protocol**

import java.util.HashMap;

import java.util.Map;

class DistanceVectorRouting {

private Map<String, Map<String, Integer>> graph; // adjacency list

public DistanceVectorRouting() {

graph = new HashMap<>();

}

public void addEdge(String source, String destination, int cost) {

graph.putIfAbsent(source, new HashMap<>());

graph.putIfAbsent(destination, new HashMap<>());

graph.get(source).put(destination, cost);

graph.get(destination).put(source, cost); // Assuming undirected graph

}

public void computeShortestPaths(String start) {

Map<String, Integer> distances = new HashMap<>();

Map<String, String> previous = new HashMap<>();

for (String node : graph.keySet()) {

distances.put(node, Integer.MAX\_VALUE);

previous.put(node, null);

}

distances.put(start, 0);

boolean updated;

do {

updated = false;

for (String u : graph.keySet()) {

for (Map.Entry<String, Integer> neighbor : graph.get(u).entrySet()) {

String v = neighbor.getKey();

int cost = neighbor.getValue();

if (distances.get(u) + cost < distances.get(v)) {

distances.put(v, distances.get(u) + cost);

previous.put(v, u);

updated = true;

}

}

}

} while (updated);

printPaths(start, distances, previous);

}

private void printPaths(String start, Map<String, Integer> distances, Map<String, String> previous) {

System.out.println("Shortest paths from " + start + ":");

for (String node : distances.keySet()) {

if (node.equals(start)) continue;

System.out.print("To " + node + " (cost: " + distances.get(node) + "): ");

printPath(previous, node);

System.out.println();

}

}

private void printPath(Map<String, String> previous, String node) {

if (node == null) return;

printPath(previous, previous.get(node));

System.out.print(node + " ");

}

public static void main(String[] args) {

DistanceVectorRouting dvRouting = new DistanceVectorRouting();

dvRouting.addEdge("A", "B", 1);

dvRouting.addEdge("A", "C", 4);

dvRouting.addEdge("B", "C", 2);

dvRouting.addEdge("B", "D", 5);

dvRouting.addEdge("C", "D", 1);

dvRouting.computeShortestPaths("A");

}

}

**Output:**

Shortest paths from A:

To B (cost: 1): A B

To C (cost: 3): A B C

To D (cost: 4): A B C D

**Link State Routing Protocol**

import java.util.\*;

class LinkStateRouting {

private Map<String, Map<String, Integer>> graph;

public LinkStateRouting() {

graph = new HashMap<>();

}

public void addEdge(String source, String destination, int cost) {

graph.putIfAbsent(source, new HashMap<>());

graph.putIfAbsent(destination, new HashMap<>());

graph.get(source).put(destination, cost);

graph.get(destination).put(source, cost); // Assuming undirected graph

}

public void computeShortestPaths(String start) {

Map<String, Integer> distances = new HashMap<>();

Set<String> visited = new HashSet<>();

PriorityQueue<Node> pq = new PriorityQueue<>(Comparator.comparingInt(n -> n.cost));

for (String node : graph.keySet()) {

distances.put(node, Integer.MAX\_VALUE);

}

distances.put(start, 0);

pq.offer(new Node(start, 0));

while (!pq.isEmpty()) {

Node current = pq.poll();

if (visited.contains(current.name)) continue;

visited.add(current.name);

for (Map.Entry<String, Integer> neighbor : graph.get(current.name).entrySet()) {

String neighborNode = neighbor.getKey();

int newDist = distances.get(current.name) + neighbor.getValue();

if (newDist < distances.get(neighborNode)) {

distances.put(neighborNode, newDist);

pq.offer(new Node(neighborNode, newDist));

}

}

}

printPaths(start, distances);

}

private void printPaths(String start, Map<String, Integer> distances) {

System.out.println("Shortest paths from " + start + ":");

for (String node : distances.keySet()) {

if (node.equals(start)) continue;

System.out.println("To " + node + " (cost: " + distances.get(node) + ")");

}

}

private static class Node {

String name;

int cost;

Node(String name, int cost) {

this.name = name;

this.cost = cost;

}

}

public static void main(String[] args) {

LinkStateRouting lsRouting = new LinkStateRouting();

lsRouting.addEdge("A", "B", 1);

lsRouting.addEdge("A", "C", 4);

lsRouting.addEdge("B", "C", 2);

lsRouting.addEdge("B", "D", 5);

lsRouting.addEdge("C", "D", 1);

lsRouting.computeShortestPaths("A");

}

}

Output:

Shortest paths from A:

To B (cost: 1)

To C (cost: 3)

To D (cost: 4)