~\OneDrive - VietNam National University - HCM INTERNATIONAL UNIVERSITY\Desktop\DSA\DSA LAB NEW\Lab 8 Graph\ITITSB22029_DoMinhDuy_Lab8\Problem 9\MapApp.java

```java
import java.util.*;

// Node class representing a vertex in the graph
class Node {
    String name;

    Node(String name) {
        this.name = name;
    }
}

// Edge class representing a weighted edge between two nodes
class Edge {
    Node destination;
    int weight;

    Edge(Node destination, int weight) {
        this.destination = destination;
        this.weight = weight;
    }
}

// Graph class representing an undirected, weighted graph
class Graph {
    private Map<Node, List<Edge>> graph;

    Graph() {
        graph = new HashMap<>();
    }

    void addNode(String name) {
        graph.putIfAbsent(new Node(name), new ArrayList<>());
    }

    void addEdge(String source, String destination, int weight) {
        Node srcNode = findNode(source);
        Node destNode = findNode(destination);

        if (srcNode == null || destNode == null)
            return;

        graph.get(srcNode).add(new Edge(destNode, weight));
        graph.get(destNode).add(new Edge(srcNode, weight));
    }

    List<Edge> getEdges(String name) {
        Node node = findNode(name);
        return node != null ? graph.get(node) : new ArrayList<>();
    }

    Node findNode(String name) {
```

```
        for (Node node : graph.keySet()) {
            if (node.name.equals(name))
                return node;
        }
        return null;
    }

    Map<Node, List<Edge>> getGraph() {
        return graph;
    }
}

// Helper class for Dijkstra's priority queue
class NodeDistance {
    Node node;
    int distance;

    NodeDistance(Node node, int distance) {
        this.node = node;
        this.distance = distance;
    }
}

// Main application class
public class MapApp {
    public static void main(String[] args) {
        Graph graph = new Graph();

        // Create nodes
        for (char c = 'A'; c <= 'K'; c++)
            graph.addNode(String.valueOf(c));
        graph.addNode("2");
        graph.addNode("G");
        graph.addNode("I");
        graph.addNode("L");
        graph.addNode("J");

        // Create edges (based on the provided adjacency list)
        graph.addEdge("A", "B", 6);
        graph.addEdge("A", "2", 10);
        graph.addEdge("2", "B", 12);
        graph.addEdge("2", "C", 12);
        graph.addEdge("2", "F", 8);
        graph.addEdge("2", "G", 16);
        graph.addEdge("G", "I", 8);
        graph.addEdge("B", "C", 11);
        graph.addEdge("B", "D", 14);
        graph.addEdge("C", "F", 3);
        graph.addEdge("C", "E", 6);
        graph.addEdge("F", "H", 16);
        graph.addEdge("F", "I", 6);
        graph.addEdge("I", "L", 17);
        graph.addEdge("I", "H", 13);
        graph.addEdge("D", "E", 4);
```

```
        graph.addEdge("D", "H", 6);
        graph.addEdge("D", "K", 15);
        graph.addEdge("E", "H", 12);
        graph.addEdge("H", "K", 12);
        graph.addEdge("H", "L", 18);
        graph.addEdge("L", "J", 20);
        graph.addEdge("K", "J", 9);

        // Task 4: Find paths from A to K
        System.out.println("Task 4: Paths from A to K");
        findPathsAndCosts(graph, "A", "K");

        // Task 5: Dijkstra's algorithm
        System.out.println("\nTask 5: Dijkstra's Algorithm");
        System.out.println("Shortest path from A to J:");
        findShortestPath(graph, "A", "J");

        System.out.println("\nShortest path from B to L:");
        findShortestPath(graph, "B", "L");
    }

    // Task 4: Find all paths from start to end using DFS
    public static void findPathsAndCosts(Graph graph, String start, String end) {
        Node startNode = graph.findNode(start);
        Node endNode = graph.findNode(end);

        if (startNode == null || endNode == null)
            return;

        List<List<String>> allPaths = new ArrayList<>();
        findPathsHelper(graph, startNode, endNode, new ArrayList<>(), allPaths);

        System.out.println("Number of paths from " + start + " to " + end + ": " +
allPaths.size());

        int minNodes = Integer.MAX_VALUE, maxNodes = Integer.MIN_VALUE;
        List<String> shortestPath = null, longestPath = null;
        int shortestCost = Integer.MAX_VALUE, longestCost = Integer.MIN_VALUE;

        for (List<String> path : allPaths) {
            int cost = calculatePathCost(graph, path);
            if (path.size() < minNodes) {
                minNodes = path.size();
                shortestPath = path;
                shortestCost = cost;
            }
            if (path.size() > maxNodes) {
                maxNodes = path.size();
                longestPath = path;
                longestCost = cost;
            }
        }

```

```
        System.out.println("Path with smallest nodes: " + shortestPath + " (Cost: " + shortestCost + ")");
        System.out.println("Path with largest nodes: " + longestPath + " (Cost: " + longestCost + ")");
    }

    private static void findPathsHelper(Graph graph, Node current, Node end, List<String> path,
            List<List<String>> allPaths) {
        path.add(current.name);

        if (current.name.equals(end.name)) {
            allPaths.add(new ArrayList<>(path));
        } else {
            for (Edge edge : graph.getEdges(current.name)) {
                if (!path.contains(edge.destination.name)) {
                    findPathsHelper(graph, edge.destination, end, path, allPaths);
                }
            }
        }

        path.remove(path.size() - 1);
    }

    private static int calculatePathCost(Graph graph, List<String> path) {
        int totalCost = 0;
        for (int i = 0; i < path.size() - 1; i++) {
            Node current = graph.findNode(path.get(i));
            Node next = graph.findNode(path.get(i + 1));
            for (Edge edge : graph.getEdges(current.name)) {
                if (edge.destination.equals(next)) {
                    totalCost += edge.weight;
                    break;
                }
            }
        }
        return totalCost;
    }

    // Task 5: Find shortest path using Dijkstra's Algorithm
    public static void findShortestPath(Graph graph, String start, String end) {
        Map<Node, Integer> distances = new HashMap<>();
        Map<Node, Node> previous = new HashMap<>();
        PriorityQueue<NodeDistance> pq = new PriorityQueue<>(Comparator.comparingInt(nd -> nd.distance));

        Node startNode = graph.findNode(start);
        Node endNode = graph.findNode(end);

        if (startNode == null || endNode == null)
            return;

        for (Node node : graph.getGraph().keySet()) {
            distances.put(node, Integer.MAX_VALUE);
```

```
            previous.put(node, null);
        }

        distances.put(startNode, 0);
        pq.add(new NodeDistance(startNode, 0));

        while (!pq.isEmpty()) {
            NodeDistance current = pq.poll();
            Node currentNode = current.node;

            for (Edge edge : graph.getEdges(currentNode.name)) {
                int newDist = distances.get(currentNode) + edge.weight;
                if (newDist < distances.get(edge.destination)) {
                    distances.put(edge.destination, newDist);
                    previous.put(edge.destination, currentNode);
                    pq.add(new NodeDistance(edge.destination, newDist));
                }
            }
        }

        printPathWithCost(endNode, previous, distances.get(endNode));
    }

    private static void printPathWithCost(Node endNode, Map<Node, Node> previous, int cost)
{
        List<String> path = new ArrayList<>();
        Node current = endNode;

        while (current != null) {
            path.add(current.name);
            current = previous.get(current);
        }

        Collections.reverse(path);

        System.out.println("Shortest Path: " + path + " (Cost: " + cost + ")");
    }
}

```