1.A

import java.util.\*;

public class DijkstraAlgorithm {

static class Node implements Comparable<Node> {

int id;

int distance;

public Node(int id, int distance) {

this.id = id;

this.distance = distance;

}

public int compareTo(Node other) {

return Integer.compare(this.distance, other.distance);

}

}

public static Map<Integer, Integer> dijkstra(Map<Integer, Map<Integer, Integer>> graph, int start) {

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

PriorityQueue<Node> pq = new PriorityQueue<>();

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

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

}

distances.put(start, 0);

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

while (!pq.isEmpty()) {

Node current = pq.poll();

int currentNode = current.id;

int currentDistance = current.distance;

if (currentDistance > distances.get(currentNode)) {

continue;

}

for (Map.Entry<Integer, Integer> neighborEntry : graph.get(currentNode).entrySet()) {

int neighbor = neighborEntry.getKey();

int weight = neighborEntry.getValue();

int distance = currentDistance + weight;

if (distance < distances.get(neighbor)) {

distances.put(neighbor, distance);

pq.add(new Node(neighbor, distance));

}

}

}

return distances;

}

public static void main(String[] args) {

Map<Integer, Map<Integer, Integer>> graph = new HashMap<>();

graph.put(0, new HashMap<>());

graph.get(0).put(1, 5);

graph.get(0).put(2, 3);

graph.put(1, new HashMap<>());

graph.get(1).put(0, 5);

graph.get(1).put(2, 1);

graph.get(1).put(3, 3);

graph.put(2, new HashMap<>());

graph.get(2).put(0, 3);

graph.get(2).put(1, 1);

graph.get(2).put(3, 2);

graph.put(3, new HashMap<>());

graph.get(3).put(1, 3);

graph.get(3).put(2, 2);

int startNode = 0;

Map<Integer, Integer> shortestDistances = dijkstra(graph, startNode);

System.out.println("Shortest distances from node " + startNode + " to other nodes:");

for (Map.Entry<Integer, Integer> entry : shortestDistances.entrySet()) {

System.out.println("Node " + entry.getKey() + ": " + entry.getValue());

}

}

}

2.A

import java.util.\*;

public class KruskalAlgorithm {

static class Edge implements Comparable<Edge> {

int source;

int destination;

int weight;

public Edge(int source, int destination, int weight) {

this.source = source;

this.destination = destination;

this.weight = weight;

}

public int compareTo(Edge other) {

return Integer.compare(this.weight, other.weight);

}

}

static class DisjointSet {

int[] parent;

public DisjointSet(int size) {

parent = new int[size];

for (int i = 0; i < size; i++) {

parent[i] = i;

}

}

public int find(int x) {

if (parent[x] != x) {

parent[x] = find(parent[x]);

}

return parent[x];

}

public void union(int x, int y) {

int rootX = find(x);

int rootY = find(y);

parent[rootX] = rootY;

}

}

public static List<Edge> kruskal(List<Edge> edges, int numVertices) {

List<Edge> minimumSpanningTree = new ArrayList<>();

Collections.sort(edges); // Sort edges by weight

DisjointSet disjointSet = new DisjointSet(numVertices);

for (Edge edge : edges) {

int sourceRoot = disjointSet.find(edge.source);

int destinationRoot = disjointSet.find(edge.destination);

if (sourceRoot != destinationRoot) {

minimumSpanningTree.add(edge);

disjointSet.union(sourceRoot, destinationRoot);

}

}

return minimumSpanningTree;

}

public static void main(String[] args) {

List<Edge> edges = new ArrayList<>();

edges.add(new Edge(0, 1, 5));

edges.add(new Edge(0, 2, 3));

edges.add(new Edge(1, 2, 1));

edges.add(new Edge(1, 3, 3));

edges.add(new Edge(2, 3, 2));

int numVertices = 4;

List<Edge> minimumSpanningTree = kruskal(edges, numVertices);

System.out.println("Minimum Spanning Tree edges:");

for (Edge edge : minimumSpanningTree) {

System.out.println(edge.source + " - " + edge.destination + ": " + edge.weight);

}

}

}

3.A

import java.util.\*;

public class UnionFindWithCycleDetection {

static class Edge {

int source;

int destination;

public Edge(int source, int destination) {

this.source = source;

this.destination = destination;

}

}

static class UnionFind {

int[] parent;

public UnionFind(int size) {

parent = new int[size];

for (int i = 0; i < size; i++) {

parent[i] = i;

}

}

public int find(int x) {

if (parent[x] != x) {

parent[x] = find(parent[x]);

}

return parent[x];

}

public void union(int x, int y) {

int rootX = find(x);

int rootY = find(y);

parent[rootX] = rootY;

}

}

public static boolean hasCycle(List<Edge> edges, int numVertices) {

UnionFind disjointSet = new UnionFind(numVertices);

for (Edge edge : edges) {

int sourceRoot = disjointSet.find(edge.source);

int destinationRoot = disjointSet.find(edge.destination);

if (sourceRoot == destinationRoot) {

return true;

}

disjointSet.union(sourceRoot, destinationRoot);

}

return false;

}

public static void main(String[] args) {

List<Edge> edges = new ArrayList<>();

edges.add(new Edge(0, 1));

edges.add(new Edge(0, 2));

edges.add(new Edge(1, 2));

edges.add(new Edge(2, 3));

int numVertices = 4;

boolean hasCycle = hasCycle(edges, numVertices);

if (hasCycle) {

System.out.println("The graph contains a cycle.");

} else {

System.out.println("The graph does not contain a cycle.");

}

}

}