Graph Algorithms - Assignment

Task 1: Dijkstra's Shortest Path Finder

Code Dijkstra's algorithm to find the shortest path from a start node to every other node in a weighted graph with positive weights.

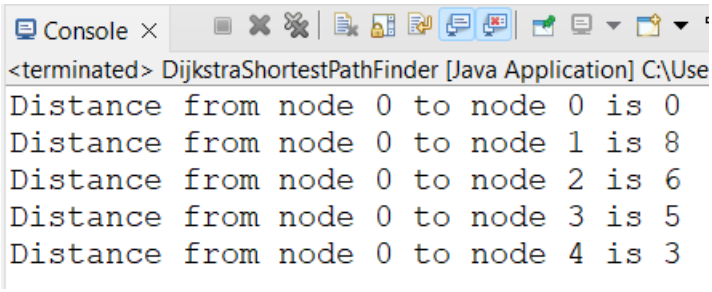
Code:

```
package dijkstra;
import java.util.*;

class Node implements Comparable<Node> {
int vertex;
int weight;
Node(int vertex, int weight) {
this.vertex = vertex;
this.weight = weight;
}

@Override
public int compareTo(Node other) {
return this.weight - other.weight;
}
}

public class DijkstraShortestPathFinder {
private int V;
private List<List<Node>> adj;

public DijkstraShortestPathFinder(int V) {
this.V = V;
adj = new ArrayList<>(V);
for (int i = 0; i < V; i++) {
adj.add(new ArrayList<>());
```

```
}
}

public void addEdge(int u, int v, int weight) {
adj.get(u).add(new Node(v, weight));
adj.get(v).add(new Node(u, weight));
}

public void dijkstra(int start) {
PriorityQueue<Node> pq = new PriorityQueue<>();
int[] dist = new int[V];
Arrays.fill(dist, Integer.MAX_VALUE);
dist[start] = 0;
pq.add(new Node(start, 0));

while (!pq.isEmpty()) {
Node node = pq.poll();
int u = node.vertex;
for (Node neighbor : adj.get(u)) {
int v = neighbor.vertex;
int weight = neighbor.weight;
if (dist[u] + weight < dist[v]) {
dist[v] = dist[u] + weight;
pq.add(new Node(v, dist[v]));
}
}
}
for (int i = 0; i < V; i++) {
System.out.println("Distance from node " + start + " to node " + i + " is " + dist[i]);
}
}

public static void main(String[] args) {
int V = 5;
```

```
DijkstraShortestPathFinder graph = new DijkstraShortestPathFinder(V);
graph.addEdge(0, 1, 9);
graph.addEdge(0, 2, 6);
graph.addEdge(0, 3, 5);
graph.addEdge(0, 4, 3);
graph.addEdge(2, 1, 2);
graph.addEdge(2, 3, 4);

graph.dijkstra(0);
}
}
```

Output:



Task 2: Kruskal's Algorithm for MST

Implement Kruskal's algorithm to find the minimum spanning tree of a given connected, undirected graph with non-negative edge weights.

Code:

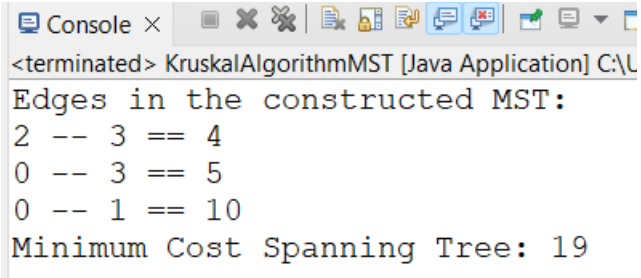
```
package kruskal;
import java.util.*;

class Edge implements Comparable<Edge> {
int src, dest, weight;

public Edge(int src, int dest, int weight) {
this.src = src;
this.dest = dest;
```

```
this.weight = weight;
}

@Override
public int compareTo(Edge other) {
return this.weight - other.weight;
}
}

class Subset {
int parent, rank;
}

public class KruskalAlgorithmMST {
int V, E;
Edge[] edges;

public KruskalAlgorithmMST(int v, int e) {
V = v;
E = e;
edges = new Edge[E];
}

int find(Subset[] subsets, int i) {
if (subsets[i].parent != i) {
subsets[i].parent = find(subsets, subsets[i].parent);
}
return subsets[i].parent;
}

void union(Subset[] subsets, int x, int y) {
int xroot = find(subsets, x);
int yroot = find(subsets, y);
if (subsets[xroot].rank < subsets[yroot].rank) {
```

```
subsets[xroot].parent = yroot;
} else if (subsets[xroot].rank > subsets[yroot].rank) {
subsets[yroot].parent = xroot;
} else {
subsets[yroot].parent = xroot;
subsets[xroot].rank++;
}
}

void kruskalMST() {
Edge[] result = new Edge[V];
int e = 0;
int i = 0;
for (i = 0; i < V; ++i) {
result[i] = new Edge(0, 0, 0);
}
Arrays.sort(edges);
Subset[] subsets = new Subset[V];
for (i = 0; i < V; ++i) {
subsets[i] = new Subset();
}
for (int v = 0; v < V; ++v) {
subsets[v].parent = v;
subsets[v].rank = 0;
}
i = 0;
while (e < V - 1) {
Edge next_edge = edges[i++];
int x = find(subsets, next_edge.src);
int y = find(subsets, next_edge.dest);
if (x != y) {
result[e++] = next_edge;
union(subsets, x, y);
}
```

```
}
System.out.println("Edges in the constructed MST:");
int minimumCost = 0;
for (i = 0; i < e; ++i) {
System.out.println(result[i].src + " -- " + result[i].dest + " == " + result[i].weight);
minimumCost += result[i].weight;
}
System.out.println("Minimum Cost Spanning Tree: " + minimumCost);
}

public static void main(String[] args) {
int V = 4;
int E = 5;

KruskalAlgorithmMST graph = new KruskalAlgorithmMST(V, E);

graph.edges[0] = new Edge(0, 1, 10);
graph.edges[1] = new Edge(0, 2, 6);
graph.edges[2] = new Edge(0, 3, 5);
graph.edges[3] = new Edge(1, 3, 15);
graph.edges[4] = new Edge(2, 3, 4);

graph.kruskalMST();
}
}
```

Output:



Task 3: Union-Find for Cycle Detection

Write a Union-Find data structure with path compression. Use this data structure to detect a cycle in an undirected graph.

Code:

```
package unionfind;
import java.util.*;

class UnionFind {
private int[] parent;
private int[] rank;

public UnionFind(int size) {
parent = new int[size];
rank = new int[size];
for (int i = 0; i < size; i++) {
parent[i] = i;
rank[i] = 0;
}
}

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);
if (rootX != rootY) {
if (rank[rootX] < rank[rootY]) {
parent[rootX] = rootY;
```

```
} else if (rank[rootX] > rank[rootY]) {
parent[rootY] = rootX;
} else {
parent[rootY] = rootX;
rank[rootX]++;
}
}
}
}

public class UnionFindCycleDetection {
private int V, E;
private List<Edge> edges;

class Edge {
int src, dest;

Edge(int src, int dest) {
this.src = src;
this.dest = dest;
}
}

public UnionFindCycleDetection(int v, int e) {
V = v;
E = e;
edges = new ArrayList<>(E);
}

public void addEdge(int src, int dest) {
edges.add(new Edge(src, dest));
}

public boolean isCycle() {
```

```
UnionFind uf = new UnionFind(V);
for (Edge edge : edges) {
int x = uf.find(edge.src);
int y = uf.find(edge.dest);
if (x == y) {
return true;
}
uf.union(x, y);
}
return false;
}

public static void main(String[] args) {
int V = 3, E = 3;
UnionFindCycleDetection graph = new UnionFindCycleDetection(V, E);
graph.addEdge(0, 1);
graph.addEdge(1, 2);
graph.addEdge(0, 2);

if (graph.isCycle()) {
System.out.println("Graph contains cycle");
} else {
System.out.println("Graph doesn't contain cycle");
}
}
}
```

Output: