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.

```
import java.util.*;


class Graph {

    private final Map<String, List<Node>> nodes = new HashMap<>();


    public void addNode(String label) {

        nodes.putIfAbsent(label, new ArrayList<>());

    }


    public void addEdge(String label1, String label2, int weight) {

        nodes.get(label1).add(new Node(label2, weight));

        nodes.get(label2).add(new Node(label1, weight));

    }


    public Map<String, Integer> dijkstra(String start) {

        // Priority queue to store (distance, node) pairs

        PriorityQueue<Node> priorityQueue = new PriorityQueue<>(Comparator.comparingInt(node
-> node.distance));

        // Map to store the shortest distance to each node

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

        // Set to keep track of visited nodes

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

```java
// Initialize distances
for (String node : nodes.keySet()) {
    distances.put(node, Integer.MAX_VALUE);
}
distances.put(start, 0);

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

while (!priorityQueue.isEmpty()) {
    Node current = priorityQueue.poll();
    String currentNode = current.label;

    if (!visited.add(currentNode)) {
        continue;
    }

    for (Node neighbor : nodes.get(currentNode)) {
        if (!visited.contains(neighbor.label)) {
            int newDist = distances.get(currentNode) + neighbor.distance;
            if (newDist < distances.get(neighbor.label)) {
                distances.put(neighbor.label, newDist);
                priorityQueue.add(new Node(neighbor.label, newDist));
            }
        }
```

```
            }
        }

        return distances;
    }

    static class Node {
        String label;
        int distance;

        Node(String label, int distance) {
            this.label = label;
            this.distance = distance;
        }
    }

    public static void main(String[] args) {
        Graph graph = new Graph();
        graph.addNode("A");
        graph.addNode("B");
        graph.addNode("C");
        graph.addNode("D");

        graph.addEdge("A", "B", 1);
        graph.addEdge("A", "C", 4);
```

```java
            graph.addEdge("B", "C", 2);
            graph.addEdge("B", "D", 5);
            graph.addEdge("C", "D", 1);

            String startNode = "A";
            Map<String, Integer> shortestPaths = graph.dijkstra(startNode);
            System.out.println(shortestPaths);
        }
}
```

OUTPUT:- {A=0, B=1, C=3, D=4}

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.

```java
import java.util.*;

class Graph {
        private final List<Edge> edges = new ArrayList<>();
        private final Map<String, String> parent = new HashMap<>();
        private final Map<String, Integer> rank = new HashMap<>();

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

```java
    }

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

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

    @Override
    public String toString() {
        return src + " - " + dest + " : " + weight;
    }
}

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

private String find(String node) {
    if (!parent.get(node).equals(node)) {
        parent.put(node, find(parent.get(node)));
    }
```

```
        return parent.get(node);
    }

    private void union(String root1, String root2) {
        if (rank.get(root1) < rank.get(root2)) {
            parent.put(root1, root2);
        } else if (rank.get(root1) > rank.get(root2)) {
            parent.put(root2, root1);
        } else {
            parent.put(root2, root1);
            rank.put(root1, rank.get(root1) + 1);
        }
    }

    public List<Edge> kruskal() {
        List<Edge> mst = new ArrayList<>();
        Collections.sort(edges);

        for (Edge edge : edges) {
            parent.putIfAbsent(edge.src, edge.src);
            parent.putIfAbsent(edge.dest, edge.dest);
            rank.putIfAbsent(edge.src, 0);
            rank.putIfAbsent(edge.dest, 0);
        }
```

```
        for (Edge edge : edges) {
            String root1 = find(edge.src);
            String root2 = find(edge.dest);

            if (!root1.equals(root2)) {
                mst.add(edge);
                union(root1, root2);
            }
        }

        return mst;
    }

    public static void main(String[] args) {
        Graph graph = new Graph();
        graph.addEdge("A", "B", 4);
        graph.addEdge("A", "H", 8);
        graph.addEdge("B", "H", 11);
        graph.addEdge("B", "C", 8);
        graph.addEdge("H", "I", 7);
        graph.addEdge("H", "G", 1);
        graph.addEdge("I", "G", 6);
        graph.addEdge("I", "C", 2);
        graph.addEdge("C", "F", 4);
        graph.addEdge("C", "D", 7);
```

```
        graph.addEdge("G", "F", 2);
        graph.addEdge("D", "F", 14);
        graph.addEdge("D", "E", 9);
        graph.addEdge("E", "F", 10);

        List<Edge> mst = graph.kruskal();
        System.out.println("Minimum Spanning Tree:");
        for (Edge edge : mst) {
            System.out.println(edge);
        }
    }
}
```

OUTPUT:- Minimum Spanning Tree:

H - G : 1

G - F : 2

I - C : 2

A - B : 4

C - F : 4

C - D : 7

A - H : 8

D - E : 9

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.

```java
import java.util.*;

class UnionFind {
    private final Map<String, String> parent = new HashMap<>();
    private final Map<String, Integer> rank = new HashMap<>();

    public void add(String node) {
        if (!parent.containsKey(node)) {
            parent.put(node, node);
            rank.put(node, 0);
        }
    }

    public String find(String node) {
        if (!parent.get(node).equals(node)) {
            parent.put(node, find(parent.get(node))); // Path compression
        }
        return parent.get(node);
    }

    public void union(String node1, String node2) {
        String root1 = find(node1);
        String root2 = find(node2);

        if (!root1.equals(root2)) {
```

```
            int rank1 = rank.get(root1);
            int rank2 = rank.get(root2);

            if (rank1 > rank2) {
                parent.put(root2, root1);
            } else if (rank1 < rank2) {
                parent.put(root1, root2);
            } else {
                parent.put(root2, root1);
                rank.put(root1, rank1 + 1);
            }
        }
    }
}

class Graph {
    private final List<Edge> edges = new ArrayList<>();

    static class Edge {
        String src, dest;

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

```
}

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

public boolean hasCycle() {
    UnionFind uf = new UnionFind();

    for (Edge edge : edges) {
        uf.add(edge.src);
        uf.add(edge.dest);

        String root1 = uf.find(edge.src);
        String root2 = uf.find(edge.dest);

        if (root1.equals(root2)) {
            return true; // Cycle detected
        }

        uf.union(edge.src, edge.dest);
    }

    return false; // No cycle detected
}
```

```java
    public static void main(String[] args) {
        Graph graph = new Graph();
        graph.addEdge("A", "B");
        graph.addEdge("A", "C");
        graph.addEdge("B", "C"); // This edge creates a cycle
        graph.addEdge("C", "D");

        if (graph.hasCycle()) {
            System.out.println("Graph contains a cycle");
        } else {
            System.out.println("Graph does not contain a cycle");
        }
    }
}
```

OUTPUT:-Graph contains a cycle