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. Ans)

Code:-

```
package com.wipro.non.linear;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.PriorityQueue;
        public class Dijkstra { private HashMap<String,
            ArrayList<Edge>> adjList =
new HashMap<>();
            private HashMap<String, Integer> distance = new
HashMap<>();
            private HashMap<String, String> previous = new
HashMap<>();
            public static void main(String[] args) {
                Dijkstra myGraph = new Dijkstra();
                myGraph.addVertex("A");
                myGraph.addVertex("B");
                myGraph.addVertex("C");
                myGraph.addVertex("D");
                myGraph.addVertex("E");
                myGraph.addVertex("F");
                myGraph.addEdge("A", "B", 2);
                myGraph.addEdge("A", "D", 8);
                myGraph.addEdge("B", "D", 5);
                myGraph.addEdge("B", "E", 6);
                myGraph.addEdge("D", "E", 3);
                myGraph.addEdge("D", "F", 2);
                myGraph.addEdge("F", "E", 1);
```

```
        myGraph.addEdge("F", "C", 3);
```

```
        myGraph.addEdge("E", "C", 9);
        myGraph.startingpont("A");
        System.out.println("Shortest distance from A to C: " +
myGraph.distance.get("C"));
        System.out.println("Shortest path from A to C: " +
myGraph.getPath("C"));
    }

    private void startingpont(String startVertex) {
        PriorityQueue<String> queue = new
PriorityQueue<>((v1, v2) -> distance.get(v1) - distance.get(v2));
        distance.put(startVertex, 0);
        queue.add(startVertex);
        while (!queue.isEmpty()) {
            String currentVertex = queue.poll();
            for (Edge edge : adjList.get(currentVertex)) { int
                newDistance = distance.get(currentVertex) +
edge.weight;
                if (!distance.containsKey(edge.vertex) ||
newDistance < distance.get(edge.vertex)) {
                    distance.put(edge.vertex, newDistance);
                    previous.put(edge.vertex, currentVertex);
                    queue.add(edge.vertex);
                }
            }
        }
    }
    private String getPath(String endVertex) {
```

```java
        StringBuilder path = new StringBuilder();
        while (endVertex!= null) {
            path.insert(0, endVertex);
            endVertex = previous.get(endVertex);
            if (endVertex!= null) {
```

```java
                path.insert(0, " -> ");
            }
        }
        return path.toString();
    }
        public boolean addEdge(String vertex1, String
vertex2, int weight) {
        if (adjList.get(vertex1)!= null && adjList.get(vertex2)!=
null) {
            adjList.get(vertex1).add(new            Edge(vertex2,
weight));
            adjList.get(vertex2).add(new Edge(vertex1,
weight));
            return true;
        }
        return false;
    }
    class Edge {
        String vertex;
        int weight;
        public Edge(String vertex, int weight) {
            this.vertex = vertex;
            this.weight = weight;
        }
```

```
        }
        public boolean addVertex(String vertex) {
            if (adjList.get(vertex) == null) {
                adjList.put(vertex, new ArrayList<Edge>());
                return true;
            }
            return false;
        }
        public void printGraph() {
      System.out.println(adjList);
    }
}
```

OUTPUT:-

```
Shortest distance from A to C: 12
Shortest path from A to C: A -> B -> D -> F -> C
```

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. Ans)

Code:-

```
package com.wipro.non.linear;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.PriorityQueue;
import java.util.Set;
public class Kruskal {
   private Map<String, List<Edge>> adjList;
   public Kruskal() {
      adjList = new HashMap<>();
```

```java
}
public static void main(String[] args) {
    Kruskal myGraph = new Kruskal();
    myGraph.addVertex("A");
    myGraph.addVertex("B");
    myGraph.addVertex("C");
    myGraph.addVertex("D");
    myGraph.addVertex("E");
```

```java
    myGraph.addVertex("F");
    myGraph.addEdge("A", "C", 3);
    myGraph.addEdge("A", "B", 2);
    myGraph.addEdge("C", "E", 4);
    myGraph.addEdge("C", "B", 5);
    myGraph.addEdge("B", "D", 3);
    myGraph.addEdge("B", "E", 4);
    myGraph.addEdge("D", "E", 2);
    myGraph.addEdge("D", "F", 3);
    myGraph.addEdge("E", "F", 5);

    List<Edge> mst = myGraph.kruskalMST();
    System.out.println("Minimum Spanning Tree:");
    for (Edge edge : mst) {
        System.out.println(edge.vertex1 + " -- " + edge.weight +
" -- " + edge.vertex2);
    }
}
private void printGraph() {
    System.out.println(adjList);
}
```

```
  public List<Edge> kruskalMST() {
      List<Edge> mst = new ArrayList<>();
      PriorityQueue<Edge> pq = new PriorityQueue<>((e1, e2)
-> e1.weight - e2.weight);
      for (Map.Entry<String, List<Edge>> entry :
adjList.entrySet()) {
          for (Edge edge : entry.getValue()) {
              pq.add(edge);
          }
      }
      UnionFind uf = new UnionFind(new
ArrayList<>(adjList.keySet()));
```

```
      while (!pq.isEmpty()) {
          Edge edge = pq.poll();
          if (!uf.isConnected(edge.vertex1, edge.vertex2)) {
              mst.add(edge);
              uf.union(edge.vertex1, edge.vertex2);
          }
      }
      return mst;
  }
  public boolean addEdge(String vertex1, String vertex2, int
weight) {
      if (adjList.get(vertex1)!= null) {
          adjList.get(vertex1).add(new Edge(vertex1, vertex2,
weight));
      }
      if (adjList.get(vertex2)!= null) {
          adjList.get(vertex2).add(new Edge(vertex2, vertex1,
```

```
weight));
    }
    return true;
  }
  public boolean addVertex(String vertex) {
    if (adjList.get(vertex) == null) {
      adjList.put(vertex, new ArrayList<>());
      return true;
    }
    return false;
  }
  public static class Edge {
    String vertex1;
    String vertex2;
    int weight;
    public Edge(String vertex1, String vertex2, int weight) {
```

```
      this.vertex1 = vertex1;
      this.vertex2 = vertex2;
      this.weight = weight;
    }
  }
  public static class UnionFind {
    private Map<String, String> parent;
    private Map<String, Integer> rank;
    public UnionFind(List<String> vertices) {
      parent = new HashMap<>();
      rank = new HashMap<>();
      for (String vertex : vertices) {
        parent.put(vertex, vertex);
```

```
            rank.put(vertex, 0);
        }
    }
    public UnionFind(Set<String> vertices) {
        this(new ArrayList<>(vertices));
    }
    public boolean isConnected(String vertex1, String vertex2)
{
        return find(vertex1).equals(find(vertex2));
    }
    public void union(String vertex1, String vertex2) {
        String root1 = find(vertex1);
        String root2 = find(vertex2);
        if (root1.equals(root2)) {
            return;
        }
        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);
        }
    }
    private String find(String vertex) {
        if (!parent.get(vertex).equals(vertex)) {
            parent.put(vertex, find(parent.get(vertex)));
        }
```

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

```
package com.wipro.non.linear;
import java.util.*;
```

```
public class CycleDetect {
```

OUTPUT:-

```
Minimum Spanning Tree:
A -- 2 -- B
D -- 2 -- E
F -- 3 -- D
A -- 3 -- C
D -- 3 -- B
```

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. Ans)

Code:-

```
private List<Edge>[] adjList;
private int[] parent;
private int[] rank;
public CycleDetect(int vertices) {
    adjList = new ArrayList[vertices];

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

        adjList[i] = new ArrayList<>();

    }

    parent = new int[vertices];
    rank = new int[vertices]; for
    (int i = 0; i < vertices; i++) {
```

```
        parent[i] = i;

        rank[i] = 0;

    }

  }

  public void addEdge(int vertex1, int vertex2, int weight) {
    adjList[vertex1].add(new Edge(vertex1, vertex2,

weight));
    adjList[vertex2].add(new Edge(vertex2, vertex1,

weight));
  }

  public boolean hasCycle() {
    for (int i = 0; i < adjList.length; i++) {
      for (Edge edge : adjList[i]) {

        int x = find(edge.vertex1);

        int y = find(edge.vertex2);

        if (x == y) {

          return true;
        } union(x,
        y);
```

```
        }

    }

    return false;
```

```
}

public int find(int vertex) {
    if (parent[vertex] != vertex) {
        parent[vertex] = find(parent[vertex]);
    }

    return parent[vertex];

}

public void union(int x, int y) {
    int x_set_parent = find(x);
    int y_set_parent = find(y);
    if (rank[x_set_parent] > rank[y_set_parent]) {

        parent[y_set_parent] = x_set_parent;

     } else if (rank[x_set_parent] < rank[y_set_parent]) {

        parent[x_set_parent] = y_set_parent;
```

```
    } else {
        parent[y_set_parent] = x_set_parent;
        rank[x_set_parent]++;

    }

}

public static class Edge {
    int vertex1;

    int vertex2;
    int weight;
    public Edge(int vertex1, int vertex2, int weight) {

        this.vertex1 = vertex1;

        this.vertex2 = vertex2;

        this.weight = weight;

    }
}
public void printGraph() {
    for (int i = 0; i < adjList.length; i++) {

        System.out.println("Vertex " + i + ":");

        for (Edge edge : adjList[i]) {
```

```
            System.out.println(" -> Vertex " + edge.vertex2 + "
(weight: " + edge.weight + ")");
        }
    }
}

public static void main(String[] args) {
    CycleDetect myGraph = new CycleDetect(6);

    myGraph.addEdge(0, 1, 4);

    myGraph.addEdge(0, 2, 4);
    myGraph.addEdge(1, 3, 2);
    myGraph.addEdge(4, 5, 3);

    myGraph.addEdge(2, 3, 3);

    myGraph.addEdge(2, 5, 2);

    myGraph.addEdge(2, 4, 4);

    myGraph.addEdge(3, 4, 3);

    myGraph.addEdge(3, 5, 5);
    myGraph.addEdge(5, 4, 3);

    myGraph.printGraph();
```

```java
        if (myGraph.hasCycle()) {

            System.out.println("Graph has a cycle");

        } else {

            System.out.println("Graph does not have a cycle");
        }
    }


}
```

OUTPUT:-

```
Vertex 0:
-> Vertex 1 (weight: 4)
-> Vertex 2 (weight: 4)
Vertex 1:
-> Vertex 0 (weight: 4)
-> Vertex 3 (weight: 2)
```

```
Vertex 2:
-> Vertex 0 (weight: 4)
-> Vertex 3 (weight: 3)
-> Vertex 5 (weight: 2)
-> Vertex 4 (weight: 4)
Vertex 3:
-> Vertex 1 (weight: 2)
-> Vertex 2 (weight: 3)
-> Vertex 4 (weight: 3)
-> Vertex 5 (weight: 5)
Vertex 4:
```

```
-> Vertex 5 (weight: 3)
-> Vertex 2 (weight: 4)
-> Vertex 3 (weight: 3)
-> Vertex 5 (weight: 3)
Vertex 5:
-> Vertex 4 (weight: 3)
-> Vertex 2 (weight: 2)
-> Vertex 3 (weight: 5)
-> Vertex 4 (weight: 3)
Graph has a cycle
```