Day 9 and 10:

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.

SOLUTION :

```
package com.dsassignment_day_9_10;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;
import java.util.PriorityQueue;

public class dijsktra{
   private HashMap<String, HashMap<String,Integer>> adjList = new HashMap<>();

   private HashMap<String, String> previous = new HashMap<>();

   public static void main(String[] args) {
      dijsktra myGraph = new dijsktra();
      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", "E", 6);
      myGraph.addEdge("B", "D", 5);
```

```java
    myGraph.addEdge("E", "D", 3);
    myGraph.addEdge("E", "F", 1);
    myGraph.addEdge("E", "C", 9);
    myGraph.addEdge("D", "F", 2);
    myGraph.addEdge("F", "C", 3);

    myGraph.printGraph();

    myGraph.dijkstra("A");

    ArrayList<String> shortestPathToC = myGraph.getShortestPathTo("C");
    System.out.println("Shortest path from A to C: " + shortestPathToC);
}

private void addVertex(String vertex) {
        if (!adjList.containsKey(vertex)) {
            adjList.put(vertex, new HashMap<>());
        }
}

public boolean addEdge(String vertex1, String vertex2, int weight) {

    if (adjList.containsKey(vertex1) && adjList.containsKey(vertex2)) {

        adjList.get(vertex1).put(vertex2, weight);
        adjList.get(vertex2).put(vertex1, weight);
        return true;
    }
    return false;
}

private void printGraph() {
```

```java
        System.out.println("Graph:");
        for (Map.Entry<String, HashMap<String, Integer>> entry : adjList.entrySet()) {
            System.out.println(entry.getKey() + " -> " + entry.getValue());
        }

    }

    private void dijkstra(String start) {
        HashMap<String, Integer> distance = new HashMap<>();
        PriorityQueue<VertexDistancePair> pq = new PriorityQueue<>((pair1, pair2) ->
Integer.compare(pair1.distance, pair2.distance));

        for (String vertex : adjList.keySet()) {
            distance.put(vertex, Integer.MAX_VALUE);
            previous.put(vertex, null);
        }

        distance.put(start, 0);
        pq.offer(new VertexDistancePair(start, 0));

        while (!pq.isEmpty()) {
            VertexDistancePair currentPair = pq.poll();
            String current = currentPair.vertex;

            for (Map.Entry<String, Integer> neighborEntry : adjList.get(current).entrySet()) {
                String neighbor = neighborEntry.getKey();
                int weight = neighborEntry.getValue();
                int newDistance = distance.get(current) + weight;

                if (newDistance < distance.get(neighbor)) {
                    distance.put(neighbor, newDistance);
                    previous.put(neighbor, current);
                    pq.offer(new VertexDistancePair(neighbor, newDistance));
```

```java
                }
            }
        }
    }

    private ArrayList<String> getShortestPathTo(String destination) {
        ArrayList<String> path = new ArrayList<>();
        String current = destination;
        while (current != null) {
            path.add(0, current);
            current = previous.get(current);
        }
        return path;
    }

    private static class VertexDistancePair {
        String vertex;
        int distance;

        VertexDistancePair(String vertex, int distance) {
            this.vertex = vertex;
            this.distance = distance;
        }
    }


}
```

OUTPUT :

```
Graph:
A -> {B=2, D=8}
B -> {A=2, D=5, E=6}
C -> {E=9, F=3}
D -> {A=8, B=5, E=3, F=2}
E -> {B=6, C=9, D=3, F=1}
F -> {C=3, D=2, E=1}
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.

SOLUTION :

```
package com.dsassignment_day_9_10;

import java.util.*;

class Edge implements Comparable<Edge> {

int src, dest, weight;

Edge(int src, int dest, int weight) {

this.src = src;

this.dest = dest;

this.weight = weight;

}

public int compareTo(Edge compareEdge) {

return this.weight - compareEdge.weight;

}

}

class DisjointSet {

int[] parent, rank;

DisjointSet(int n) {
```

```java
parent = new int[n];
rank = new int[n];
for (int i = 0; i < n; i++) {
parent[i] = i;
rank[i] = 0;
}
}
int find(int x) {
if (parent[x] != x)
parent[x] = find(parent[x]);
return parent[x];
}
void union(int x, int y) {
int xRoot = find(x);
int yRoot = find(y);
if (xRoot == yRoot)
return;
if (rank[xRoot] < rank[yRoot])
parent[xRoot] = yRoot;
else if (rank[xRoot] > rank[yRoot])
parent[yRoot] = xRoot;
else {
parent[yRoot] = xRoot;
rank[xRoot]++;
}
}
}
public class KruskalMST {
```

```java
private int V, E;

private Edge[] edges;

KruskalMST(int v, int e) {

V = v;

E = e;

edges = new Edge[E];

for (int i = 0; i < e; ++i)

edges[i] = new Edge(0, 0, 0);

}

void addEdge(int e, int src, int dest, int weight) {

edges[e].src = src;

edges[e].dest = dest;

edges[e].weight = weight;

}

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);

DisjointSet ds = new DisjointSet(V);

i = 0;

while (e < V - 1) {

Edge nextEdge = edges[i++];

int x = ds.find(nextEdge.src);

int y = ds.find(nextEdge.dest);

if (x != y) {
```

```java
result[e++] = nextEdge;

ds.union(x, y);

}

}

System.out.println("Edges in the minimum spanning tree:");

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 of the spanning tree: " + minimumCost);

}

public static void main(String[] args) {

int V = 4;

int E = 5;

KruskalMST graph = new KruskalMST(V, E);

graph.addEdge(0, 0, 1, 10);

graph.addEdge(1, 0, 2, 6);

graph.addEdge(2, 0, 3, 5);

graph.addEdge(3, 1, 3, 15);

graph.addEdge(4, 2, 3, 4);

graph.kruskalMST();

}

}
```

OUTPUT :

```
Edges in the minimum spanning tree:
2 - 3: 4
0 - 3: 5
0 - 1: 10
Minimum cost of the spanning tree: 19
```

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.

SOLUTION :

```java
package com.dsassignment_day_9_10;

import java.util.Arrays;

class UnionFind {

int[] parent;

int[] rank;

UnionFind(int n) {

parent = new int[n];

rank = new int[n];

Arrays.fill(rank, 1);

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

parent[i] =i;

}

}

int find(int i) {

if (parent[i] != i) {

parent[i] = find(parent[i]);
```

```
}
return parent[i];
}
void union(int x, int y) {
int rootX = find(x);
int rootY = find(y);
if (rootX != rootY) {
if (rank[rootX] < rank[rootY]) { // 1<2
parent[rootX] = rootY;
} else if (rank[rootX] > rank[rootY]) {
parent[rootY] = rootX;
} else {
parent[rootY] = rootX;
rank[rootX]++;
}
}
}
}
class Graph {
int V, E;
Edge[] edges;
class Edge {
int src, dest;
}
Graph(int v, int e) {
this.V = v;
this.E = e;
this.edges = new Edge[E];
```

```
for (int i = 0; i < e; i++) {

edges[i] = new Edge();

System.out.println(edges[i].src + " -- " + edges[i].dest);

}

}

public boolean isCycleFound(Graph graph) {

UnionFind uf = new UnionFind(V);

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

int x = find(uf, graph.edges[i].src);

int y = find(uf, graph.edges[i].dest);

if(x==y) {

return true;

}

uf.union(x, y);

}

return false;

}

private int find(UnionFind uf, int i) {

return uf.find(i);

}

}

public class CycleDetect {

public static void main(String[] args) {

//int V = 3, E = 3;

int V = 3, E = 2;

Graph graph = new Graph(V, E);

graph.edges[0].src = 0;

graph.edges[0].dest = 1;
```

```java
graph.edges[1].src = 1;

graph.edges[1].dest = 2;

//graph.edges[2].src = 0;

//graph.edges[2].dest = 2;

System.out.println(graph.V + " -- " + graph.E);

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

System.out.println(graph.edges[i].src + " -- " + graph.edges[i].dest);

}

if(graph.isCycleFound(graph)) {

System.out.println("Cycle Found");

}else {

System.out.println("Cycle Not Found...");

}

}

}
```

OUTPUT :

```
0  -- 0
0  -- 0
3 -- 2
0  -- 1
1  -- 2
Cycle Not Found...
```