Assignment – 13

Name - Md. Firoze baba
firozebaba68@gmail.com

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.

Program:

```
import java.util.*;

public class DijkstraAlgorithm {
     private static final int INF = Integer.MAX_VALUE;
     public static void main(String[] args) {
          int[][] graph = {
                    { 0, 4, 0, 8, 0},
                    { 4, 0, 8, 11, 0},
                    { 0, 8, 7, 0, 4},
                    { 0, 7, 0, 9, 14},
                    { 0, 0, 0, 9, 10}};
          dijkstras(graph, 0);
     }

     public static void dijkstras(int[][] graph, int src) {
          int v = graph.length;
          int[] dij = new int[v];
          boolean[] visited = new boolean[v];
          Arrays.fill(dij, INF);
          dij[src] = 0;
          for (int count = 0; count < v - 1; count++) {
               int u = minDistance(dij, visited);
               visited[u] = true;
               for (int j = 0; j < v; j++) {
                    if (!visited[j] && graph[u][j] != 0 &&
                         dij[u] != INF && dij[u] + graph[u][j] <
                          dij[j]) {
                         dij[j] = dij[u] + graph[u][j];
                    }
               }
          }
          printResult(dij);
     }

     public static int minDistance(int[] dij, boolean[] visited) {
          int min = INF;
          int minindex = -1;
          for (int v = 0; v < dij.length; v++) {
               if (!visited[v] && dij[v] <= min) {
                    min = dij[v];
                    minindex = v;
```

```
                }
            }
            return minindex;
        }

        public static void printResult(int[] dij) {
            System.out.println("vertex \t distance from source: ");
            for (int i = 0; i < dij.length; i++) {
                System.out.println(i + "\t" + dij[i]);
            }
        }

}
```

Output:

```
vertex      distance from source:
0     0
1     4
2     12
3     8
4     16
```

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.

Program:

```
import java.util.Arrays;
import java.util.LinkedList;

class Edge implements Comparable<Edge>{
        int src,dest, weight;
        public Edge(int src, int dest, int weight) {
            super();
            this.src = src;
            this.dest = dest;
            this.weight = weight;
        }
        @Override
        public int compareTo(Edge e1) {
            return this.weight-e1.weight;
        }
}

public class KruskalsAlgorithm {
        private int vertex;
        private  LinkedList<Edge> edges;
        public KruskalsAlgorithm(int v) {
```

```
        vertex=v;
        edges=new LinkedList<>();
    }

    public void addEdges(int src,int dest,int weight) {
        Edge edge=new Edge(src, dest, weight);
        edges.add(edge);
    }

    private int find(int[] parent, int i) {
        if(parent[i] != i && parent[i] != -1)
            parent[i]=find(parent, parent[i]);
        return parent[i] == -1 ? i : parent[i];
    }

    private void union(int[] parent, int x,int y) {
        int xset=find(parent, x);
        int yset=find(parent, x);
        parent[xset]=yset;
    }

    public void kruskalMST() {
        Edge[] resultEdges=new Edge[vertex];
        int e=0;
        int i=0;
        edges.sort(null);
        int[] parent=new int[vertex];
        Arrays.fill(parent, -1);
        while(e<vertex-1) {
            Edge next=edges.get(i++);
            int x=find(parent, next.src);
            int y=find(parent, next.dest);
            if(x != y) {
                resultEdges[e++]=next;
                union(parent, x, y);
            }
        }
        System.out.println("Minimal Spanning Tree: ");
        int totalWeight=0;
        for(i=0;i<e;i++) {
            System.out.println(resultEdges[i].src+
                                " -
                        "+resultEdges[i].dest+": "
                                +resultEdges[i].weight);
            totalWeight +=resultEdges[i].weight;
        }
        System.out.println("Total weight: "+totalWeight);
    }

    public static void main(String[] args) {
        int v=4;
```

```
        KruskalsAlgorithm graph=new KruskalsAlgorithm(v);
        graph.addEdges(0, 1, 8);
        graph.addEdges(0, 2, 7);
        graph.addEdges(0, 3, 5);
        graph.addEdges(1, 3, 10);
        graph.addEdges(2, 3, 4);
        graph.kruskalMST();
    }
}
```

Output:

```
Minimal Spanning Tree:
2 - 3: 4
0 - 3: 5
0 - 2: 7
Total weight: 16
```

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.

Program:

```
public class UnionFind {
    private int[] parent;
    private int[] rank;
    public UnionFind(int n) {
        parent = new int[n];
        rank = new int[n];
        for (int i = 0; i < n; 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)
            return;
        if (rank[rootx] < rank[rooty]) {
            parent[rootx] = rooty;
        } else if (rank[rootx] > rank[rooty]) {
```

```
                parent[rooty] = rootx;
            } else {
                parent[rooty] = rootx;
                rank[rooty]++;
            }
        }
}



public class UnionFindAlgorithm {
        public static boolean hasCycle(int[][] edges,int n) {
                UnionFind uf=new UnionFind(n);
                for(int[] edge:edges) {
                        int u=edge[0];
                        int v=edge[1];
                        int rootu=uf.find(u);
                        int rootv=uf.find(v);
                        if(rootu == rootv) {
                                return true;
                        }
                        uf.union(rootu, rootv);
                }
                return false;
        }

        public static void main(String[] args) {
                int[][] edges= {
                                {0,1},
                                {1,2},
                                {2,3},
                                {3,0}};
                int n=4;
                boolean cycle=hasCycle(edges, n);
                if(cycle) {
                        System.out.println("Cycle detected!");
                }
                else {
                        System.out.println("Cycle not detected.");
                }
        }

}
```

Output:

```
Cycle detected!
```