**Report MDISC – Explanation US013**

**Group 11**

## Kruskal’s Algorithm

Kruskal's algorithm is widely recognized as one of the most effective methods for determining the Minimum Spanning Tree (MST) in a graph. An MST in a connected graph is a tree-like structure that connects all vertices of the graph with the lowest possible total cost. The Kruskal process is based on selecting edges with the lowest possible cost, while ensuring that these choices do not create cycles when added to the MST.

### 1. DisjointSet.java

This class represents a disjoint set data structure. It is used to manage a set of elements partitioned into a number of disjoint (non-overlapping) subsets.

It has two main methods: find and union. The find method is used to determine the parent of an element (i.e., the representative of the subset to which the element belongs). The union method is used to merge two subsets into a single subset.

This class is used in Kruskal's algorithm to check if two nodes belong to the same subset (to avoid creating a cycle) and to merge the subsets of two nodes when an edge is added to the minimum spanning tree.

```java
public class DisjointSet {
    private int[] parent;
    private int[] rank;

    public DisjointSet(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 element) {
        if (parent[element] != element) {
            parent[element] = find(parent[element]);
        }
        return parent[element];
    }

    public void union(int element1, int element2) {
        int root1 = find(element1);
        int root2 = find(element2);
        if (root1 != root2) {
            if (rank[root1] > rank[root2]) {
                parent[root2] = root1;
            } else if (rank[root1] < rank[root2]) {
                parent[root1] = root2;
            } else {
                parent[root2] = root1;
                rank[root1]++;
            }
        }
    }
```

2. **Edge.java**

This class represents an edge in a graph. It has three properties: src (the source node), dest (the destination node), and weight (the weight of the edge).

```java
public class Edge {
    int src, dest, weight;

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

3. **IrrigationSystem.java**

This class represents an irrigation system, modeled as a graph where the nodes represent points in the system and the edges represent connections between the points.

The importCsv method is used to import a graph from a CSV file. The graph is represented as a 2D array where the cell at the i-th row and j-th column represents the weight of the edge between the i-th and j-th nodes.

The sortEdges method is used to sort the edges of the graph in ascending order of their weights. This is a step in Kruskal's algorithm.

The planIrrigationSystem method implements Kruskal's algorithm to find the minimum spanning tree of the graph. It uses the DisjointSet class to keep track of the subsets of nodes and the sortEdges method to sort the edges before processing. Finally, the visualizeGraph method is used to visualize the input graph and the minimum spanning tree using the GraphStream library.

```java
public class IrrigationSystem {
    public void importCsv(String filename) {
        List<String> nodes = new ArrayList<>();
        int totalEdges = 0;
        try (BufferedReader br = new BufferedReader(new FileReader(filename))) {
            String line;
            while ((line = br.readLine()) != null && !line.isEmpty()) {
                totalEdges++; // Count each line as an edge
                String[] values = line.split( regex: ",");
                if (!nodes.contains(values[0])) {
                    nodes.add(values[0]);
                }
                if (!nodes.contains(values[1])) {
                    nodes.add(values[1]);
                }
            }
            int size = nodes.size();
            graph = new int[size][size];
            this.totalEdges = totalEdges;
        } catch (IOException e) {
            e.printStackTrace();
        }

        try (BufferedReader br = new BufferedReader(new FileReader(filename))) {
            String line;
            while ((line = br.readLine()) != null && !line.isEmpty()) {
                String[] values = line.split( regex: ",");
                int pointX = nodes.indexOf(values[0]);
                int pointY = nodes.indexOf(values[1]);
                int distance = Integer.parseInt(values[2]);
                graph[pointX][pointY] = distance;
                graph[pointY][pointX] = distance; // The graph is undirected
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
```

```
public class IrrigationSystem {
    public void planIrrigationSystem(String inputFilePath, boolean openGUI) {  4 usages  Ricardo +1
        int vertices = graph.length;
        List<Edge> edges = new ArrayList<>();
        for (int i = 0; i < vertices; i++) {
            for (int j = i + 1; j < vertices; j++) {
                if (graph[i][j] != 0) {
                    edges.add(new Edge(i, j, graph[i][j]));
                }
            }
        }

        // Sort the edges in ascending order of weight
        edges.sort(Comparator.comparingInt(edge -> edge.weight));

        DisjointSet ds = new DisjointSet(vertices);

        int[][] mst = new int[vertices][vertices];
        int totalCost = 0;

        for (Edge edge : edges) {
            int root1 = ds.find(edge.src);
            int root2 = ds.find(edge.dest);
            if (root1 != root2) {
                mst[edge.src][edge.dest] = edge.weight;
                mst[edge.dest][edge.src] = edge.weight;
                totalCost += edge.weight;
                ds.union(root1, root2);
            }
        }

        printCSV(mst, totalCost, inputFilePath);
        printMST(totalCost, inputFilePath);

        // Visualize the input graph
        visualizeGraph(graph,  title: "Input Graph", openGUI, inputFilePath);
```

**The algorithm works as follows:**

- Import the graph from a CSV file using the importCsv method.
- Sort the edges of the graph in ascending order of their weights using the sortEdges method.
- Initialize a new disjoint set with a size equal to the number of nodes in the graph.
- Initialize an empty minimum spanning tree.
- Iterate over the sorted edges. For each edge, use the find method of the disjoint set to check if the two nodes of the edge belong to the same subset. If they do, skip the edge (as adding it to the minimum spanning tree would create a cycle). If they don't, add the edge to the minimum spanning tree and use the union method of the disjoint set to merge the subsets of the two nodes.
- Continue until the minimum spanning tree contains n - 1 edges (where n is the number of nodes in the graph) or until all edges have been processed.
- Visualize the input graph and the minimum spanning tree using the visualizeGraph method.