Introduction

When analysing algorithms to determine the best approach for solving a problem, it is crucial to assess both the time and space complexity. The primary goal is to determine the performance of these algorithms independent of hardware and programming language.

The analysis involves identifying the number of primitive operations, such as arithmetic operations, comparisons, list iterations, value assignments, function returns, and function calls, and evaluating their impact on the overall performance.

This report focuses on the analysis of Kruskal's and Dijkstra's algorithms implemented in US013, US017 and US018.

Algorithm Analysis US013

Method: planIrrigationSystem

The **`planIrrigationSystem`** method constructs a Minimum Spanning Tree (MST) using Kruskal's algorithm. This involves sorting the edges and using a disjoint set to avoid cycles.

```
public void planIrrigationSystem(String inputFilePath, boolean openGUI) {
        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
        sortEdges(edges);

        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);
            }
        }
}
```

Code Analysis

Line(s) of Code	Pseudocode Operation	No. of Primitive Operations
2	Initialize *vertices* to the length of *graph*	1
3	Initialize *edges* as an empty list	1
4-9	Nested loop to iterate through all pairs of vertices	$\frac{n(n-1)}{2}$
6	Check if the weight of edge *(i, j)* is not zero	$\frac{n(n-1)}{2}$
7	Add edge *(i, j)* to *edges* list	$\frac{n(n-1)}{2}$
11	Sort the edges in ascending order of weight	$E \log E$
13	Initialize a disjoint set with *vertices*	1
15	Initialize *mst* matrix and *totalCost*	$n^2 + 1$
17-23	Loop through sorted edges	E
18-19	Find roots of the source and destination vertices	$2 \log V$
20	Check if roots are different	E
21-23	Add edge to MST, update total cost, and union the sets	$1 + 1 + \log V$

Time Complexity Analysis

Edge Collection: The nested loops iterate through all pairs of vertices, resulting in a time complexity of $O(n^2)$.

Edge Sorting: Sorting the edges has a time complexity of $O(E \log E)$, where E is the number of edges.

Union-Find Operations: Each *find* and *union* operation is approximately O(log V) due to path compression and union by rank.

Overall, the ***planIrrigationSystem*** method has a time complexity of $O(E \log E + V^2)$.

Algorithm Analysis US017

Method: dijkstra

The **`dijkstra`** method implements Dijkstra's single-source shortest path algorithm using an adjacency matrix representation.

```
public DijkstraResult dijkstra(int source) {
        int n = graph.length;
        int[] shortestDistances = new int[n];
        boolean[] added = new boolean[n];
        for (int vertexIndex = 0; vertexIndex < n; vertexIndex++) {
            shortestDistances[vertexIndex] = Integer.MAX_VALUE;
            added[vertexIndex] = false;
        }
        shortestDistances[source] = 0;
        int[] parents = new int[n];
        parents[source] = -1;
        for (int i = 1; i < n; i++) {
            int nearestVertex = -1;
            int shortestDistance = Integer.MAX_VALUE;
            for (int vertexIndex = 0; vertexIndex < n; vertexIndex++) {
                if (!added[vertexIndex] && shortestDistances[vertexIndex] < shortestDistance) {
                    nearestVertex = vertexIndex;
                    shortestDistance = shortestDistances[vertexIndex];
                }
            }
            added[nearestVertex] = true;
            for (int vertexIndex = 0; vertexIndex < n; vertexIndex++) {
                int edgeDistance = graph[nearestVertex][vertexIndex];
                if (edgeDistance > 0 && ((shortestDistance + edgeDistance) <
shortestDistances[vertexIndex])) {
                    parents[vertexIndex] = nearestVertex;
                    shortestDistances[vertexIndex] = shortestDistance + edgeDistance;
                }
            }
        }
        return new DijkstraResult(parents, shortestDistances);
}
```

Code Analysis

Line(s) of Code	Pseudocode Operation	No. of Primitive Operations
2	Initialize `n` to the length of `graph`	1
3-5	Initialize `shortestDistances` and `added` arrays	2n
6	Set the distance of the source to 0	**1**
7	Initialize `parents` array	n
8-18	Outer loop to process each vertex	n
9-14	Inner loop to find the nearest vertex	n
15-18	Inner loop to update distances	n
19	Return `DijkstraResult`	1

Time Complexity Analysis

Initialization: The initialization of arrays has a time complexity of O(n).

Main Loop: The outer loop runs (n - 1) times, and the inner loop iterates n times to find the nearest vertex, resulting in $O(n^2)$ complexity.

Relaxation: The nested loops to update the shortest distances also contribute $O(n^2)$

The overall complexity of the ***Dijkstra*** method is **O(n^2)**.

Algorithm Analysis US018

Method: findShortestPathToAP

The **`findShortestPathToAP`** method calculates the shortest path from each location to the nearest assembly point using Dijkstra's algorithm.

```
public void findShortestPathToAP(String matrixFileName) {
        if (assemblyPoints.isEmpty()) {
            System.out.println("No assembly points found");
            return;
        }
        for (int i = 0; i < locationNames.length; i++) {
            int nearestAPIndex = -1;
            int shortestDistance = Integer.MAX_VALUE;
            for (String ap : assemblyPoints) {
                int apIndex = -1;
                for (int j = 0; j < locationNames.length; j++) {
                    if (locationNames[j].equals(ap)) {
                        apIndex = j;
                        break;
                    }
                }
                if (apIndex == -1) {
                    System.out.println("AP " + ap + " not found");
                    continue;
                }
                DijkstraResult result = dijkstra(i);
                int[] shortestDistances = result.getShortestDistances();
                if (shortestDistances[apIndex] < shortestDistance) {
                    shortestDistance = shortestDistances[apIndex];
                    nearestAPIndex = apIndex;
                }
            }
            if (nearestAPIndex != -1) {
                DijkstraResult result = dijkstra(i);
                int[] parents = result.getParents();
                System.out.print("Path from " + locationNames[i] + " to " +
locationNames[nearestAPIndex] + ": ");
                printShortestPath(i, nearestAPIndex, parents);
                System.out.println(" with cost " + shortestDistance);
                 writePathsToCSV("src/main/java/pt/ipp/isep/dei/mdisc/database/input/
paths.csv",  matrixFileName);
            }
        }
}
```

Code Analysis

Line(s) of Code	Pseudocode Operation	No. of Primitive Operations
2-3	Check if *assemblyPoints* list is empty	1
4-25	Outer loop to iterate through all locations	L
6-23	Inner loop to iterate through all assembly points	$L \times A$

8-14	Nested loop to find the index of the assembly point in *locationNames*	$L \times A \times n$
16	Execute *dijkstra* method	$L \times A \times n^2$
18-19	Check and update the shortest distance to the assembly point	$L \times A \times n$
21-24	Execute *dijkstra* method and print the shortest path	$L \times n^2$
25	Write the paths to a CSV file	L

Time Complexity Analysis

Checking Assembly Points: The outer loop iterates over all locations, and the inner loop iterates over all assembly points, resulting in O(*L* x *A*), where *L* is the number of locations and *A* is the number of assembly points.

Dijkstra's Algorithm: For each location and assembly point pair, Dijkstra's algorithm is executed, contributing $O(n^2)$ complexity for each execution.

Overall, the ***findShortestPathToAP*** method has a time complexity of $O(L \times A \times n^2)$.

Conclusion

This document provides a detailed analysis of the algorithms specified in user stories US013, US017, and US018. The analysis focuses on Kruskal's algorithm for constructing a Minimum Spanning Tree in the *planIrrigationSystem* method and Dijkstra's algorithm for shortest path calculations in the *dijkstra* and *findShortestPathToAP* methods. By breaking down the time complexities of these methods and their components, the document offers valuable insights into their computational costs.

For US013, the *planIrrigationSystem* method uses Kruskal's algorithm to ensure the irrigation network is established at the lowest possible cost. The primary computational expense is the sorting of edges, which dominates the overall complexity.

For US017 and US018, Dijkstra's algorithm is used to identify the shortest paths for different purposes. In US017, it determines the shortest path from a single source, contributing to the efficiency of routing decisions. In US018, it calculates the shortest paths to the nearest assembly points for each location, aiding in emergency planning.

To conclude, this analysis offers help in understanding the efficiency and performance implications of each method in the given user stories as well in different scenarios and for different input sizes:

- **US013**: The `planIrrigationSystem` method has a complexity of **O(n log n)**.
- **US017**: The `dijkstra` method has a complexity of **$O(n^2)$**.
- **US018**: The `findShortestPathToAP` method has a complexity of **$O(n^3)$**.