US19 - Complexity Analysis

1010488 - Flávio Cruz
1211572 - David Mendonça

LEI - MDISC
2SEM - 1NB - G322

June 2024

The pseudocode along with the time complexity analysis for the procedures developed in US13, US17, and US18. The time complexity analysis is summarized in a table for each procedure, showing the individual and total complexities.

US13 - Kruskal Algorithm

```
Function findMinimumSpanningTree(Graph graph):
    edges <- list of all edges in graph
    Sort(edges)  // Sort edges by weight
    initializeUnionFind(graph.getVertices())
    result <- empty set
    For each edge in edges:
        root1 <- find(edge.from)
        root2 <- find(edge.to)
        If root1 != root2:
            result.add(edge)
            union(root1, root2)
    Return result

Function Sort(List edges):
    For i from 0 to size(edges) - 1:
        For j from i + 1 to size(edges):
            If edges[i].compare(edges[j]) > 0:
                swap(edges[i], edges[j])

Function initializeUnionFind(Set vertices):
    For each vertex in vertices:
        parent[vertex] <- vertex
        rank[vertex] <- 0

Function find(Vertex vertex):
    If parent[vertex] != vertex:
        parent[vertex] <- find(parent[vertex])  // Path compression
    Return parent[vertex]

Function union(Root root1, Root root2):
    If rank[root1] > rank[root2]:
        parent[root2] <- root1
    Else If rank[root1] < rank[root2]:
```

```
        parent[root1] <- root2
    Else If root1 != root2:
        parent[root2] <- root1
        rank[root1] <- rank[root1] + 1
```

Line	Operation	Complexity
1	Assignment of edge list	O(n)
2	Sort edges (bubble sort)	O(n^2)
3	Initialize union-find	O(n)
4	Initialize result set	O(1)
5-9	Loop over edges and union-find operations	O(n log n)
10	Return result	O(1)

Table 1: Complexity Analysis of Kruskal Algorithm

US17 - Dijkstra Algorithm

```
Class EmergencyPathFinder:
    weights: matrix of integers
    points: array of strings
    assemblyPoints: list of integers

    Constructor(weights, points, assemblyPoints):
        weights <- weights
        points <- points
        assemblyPoints <- assemblyPoints

    Method dijkstra(startIdx, endIdx):
        n <- size of weights matrix
        dist <- array of size n initialized with INFINITY
        visited <- boolean array of size n initialized with FALSE
        prev <- integer array of size n initialized with -1
        dist[startIdx] <- 0
        priorityQueue <- priority queue comparing based on dist
        add startIdx to priorityQueue

        While priorityQueue is not empty:
            u <- extract minimum from priorityQueue
            If u == endIdx, break
            If visited[u], continue
            visited[u] <- TRUE
            For each v from 0 to n-1:
                If weights[u][v] > 0 and not visited[v]:
                    newDist <- dist[u] + weights[u][v]
                    If newDist < dist[v]:
                        dist[v] <- newDist
                        prev[v] <- u
                        add v to priorityQueue
        path <- empty list
        For at <- endIdx while at != -1:
            add at to path
            at <- prev[at]
        reverse path
        return path

    Method calculateAllPaths():
```

```
paths <- empty list
For i from 0 to size of points - 1:
    If i not in assemblyPoints:
        closestAPIdx <- -1
        minDist <- INFINITY
        For each apIdx in assemblyPoints:
            path <- dijkstra(i, apIdx)
            dist <- size of path - 1
            If dist < minDist:
                minDist <- dist
                closestAPIdx <- apIdx
        add dijkstra(i, closestAPIdx) to paths
    Else:
        add empty list to paths
return paths
```

Line	Operation	Complexity
1	Constructor initialization	O(1)
2-4	Initialization of arrays and queue	O(n)
5-9	While loop with priority queue	O(n log n + n)
10-13	Path reconstruction	O(n)
14	Return path	O(1)
Total (dijkstra)		**O(n log n + n)**
15-25	calculateAllPaths (calls Dijkstra)	O(n^2 log n + 2n)
Total (calculateAllPaths)		**O(n^2 log n + 2n)**

Table 2: Complexity Analysis of Dijkstra Algorithm and calculateAllPaths

US18 - Path to the Best Assembly Point

```
List of Integers assemblyPoints <- empty list
For i from 0 to size of points - 1:
    If points[i] starts with "AP":
        add i to assemblyPoints

epf <- new EmergencyPathFinder(weights, points, assemblyPoints)
List of Strings outputPaths <- empty list

For i from 0 to size of points - 1:
    If i not in assemblyPoints:
        closestAPIdx <- -1
        minDist <- INFINITY
        For each apIdx in assemblyPoints:
            path <- epf.dijkstra(i, apIdx)
            dist <- size of path - 1
            If dist < minDist:
                minDist <- dist
                closestAPIdx <- apIdx
        path <- epf.dijkstra(i, closestAPIdx)
        pathString <- formatPath(path, points)
        indexPathString <- formatIndexPath(path)
        cost <- calculatePathCost(path, weights)
        add indexPathString + "; Cost: " + cost + "\n" + pathString + "; Total cost: " +
```

Line	Operation	Complexity
1-4	Initialize assemblyPoints list	O(n)
5-9	Loop over points and find closest assembly point (calls Dijkstra)	$O(n^2 \log n + 2n)$
10-13	Format and calculate path cost	O(n)
14	Add formatted path to outputPaths	O(1)
Total		$\mathbf{O(n^2 \log n + 2n)}$

Table 3: Complexity Analysis of Path to the Best Assembly Point

Summary

The worst-case time complexity analysis for each procedure indicates that US13's complexity is dominated by the sorting algorithm used (bubble sort, $O(n^2)$), whereas US17 and US18 are dominated by the Dijkstra algorithm with a priority queue O(n log n) and its multiple calls.

Procedure	Worst-Case Time Complexity
US13 - Kruskal Algorithm	$O(n^2)$
US17 - Dijkstra Algorithm	O(n log n)
US17 - All Paths	$O(n^2)$
US18 - Path to the Best Assembly Point	$O(n^2)$

Table 4: Summary of Worst-Case Time Complexities