

Worst-Case Time Complexity Analysis

G131 – 1DM

1230839 – Emanuel Almeida

1230564 - Francisco Santos

Introduction

This document presents the theoretical framework and the worst-case time complexity analysis of the developed algorithms in US13, US17, and US18. All algorithms are presented in pseudo-code, along with the complexity analysis.

Algorithm Complexity

Time complexity of an algorithm measures the time taken for execution as a function of the input size. In worst-case time complexity analysis, we use Big-O notation to express an upper bound on the execution time as the input size grows.

Big-O Notation

Big-O notation characterizes the execution time or space usage of an algorithm in terms of the input size N. For example, an algorithm with complexity O(n) is linear, meaning the execution time grows linearly with the input size.

Algorithms and Complexity Analysis

US 13

- **removeCycles()**

Explanation

This algorithm iterates over each edge in the graph to check if it forms a cycle (i.e., if both vertices of the edge are the same). If a cycle is found, the edge is removed. The outer loop runs $O(N)$ times, and removing an edge can take $O(N)$ time in the worst case if implemented using an array list.

```
Input: graph (list of edges)

For each edge (u, v) in graph do      // O(N)
    If u = v then
        Remove edge (u, v) from graph  // O(N)
    End If
End For
End Algorithm
```

Worst-case Time Complexity: $O(N)$, where N is the number of edges in the graph.

- **OrderEdgesByDistance()**

Explanation

This algorithm orders edges by their distance using a simple bubble sort mechanism. The nested loops both run $O(N)$ times, leading to a quadratic complexity.

```

Input: graph (list of edges)

For i from 0 to size of graph - 1 do //  $O(N)$ 
    For j from i + 1 to size of graph do //  $O(N)$ 
        If graph[i] > graph[j] then
            Swap graph[i] and graph[j]
        End If
    End For
End For
End Algorithm

```

Worst-case Time Complexity: $O(N^2)$, where N is the number of edges in the graph.

- **removeParallelEdges()**

Explanation

This algorithm first orders the edges by distance and then removes any parallel edges. The nested loops contribute to a cubic complexity due to the additional $O(N)$ time required to remove an edge.

Input: graph (list of edges)

orderEdgesByDistance() // O(N²)

For i from 0 to size of graph - 1 do // O(N)

For j from size of graph - 1 down to i + 1 do // O(N)

If graph[i] = graph[j] then

Remove edge at position j from graph // O(N)

End If

End For

End For

End Algorithm

Worst-case Time Complexity: $O(N^3)$, where N is the number of edges in the graph.

- **GetVertices()**

Explanation

This algorithm extracts all unique vertices from the graph. Checking for the presence of a vertex and adding a vertex to the list each take $O(N)$ time, leading to quadratic complexity when combined with the outer loop.

Input: None

vertices = new empty list

For each edge in graph do // O(N)

If vertices does not contain PointX then

Add PointX to vertices // O(N)

End If

If vertices does not contain PointY then

Add PointY to vertices // O(N)

End If

End For

Return vertices

End Algorithm

Worst-case Time Complexity: $O(N^2)$, where N is the number of edges in the graph.

- **CreateAndFillContainers()**

Explanation

This algorithm creates a container for each vertex and adds it to the list of containers.

Input: containers (list of lists of vertices)

vertices = getVertices() // O(N²)

For each vertex in vertices do // O(N)

container = new list

Add vertex to container // O(N)

Add container to containers // O(N)

End For

End Algorithm

Worst-case Time Complexity: $O(N^2)$, where N is the number of edges in the graph.

- **GetContainerWithVertex()**

Explanation

This algorithm searches for a container that contains a specific vertex. The nested loops contribute to quadratic complexity as each container must be checked.

Input: vertex (vertex to search for), containers (list of lists of vertices)

For each container in containers do // O(N)

If container contains vertex then // O(N)

Return container //

End If

End For

Return null

End Algorithm

Worst-case Time Complexity: $O(N^2)$, where N is the number of edges in the graph.

- **treeBuilding()**

Explanation

This algorithm builds a tree from the graph by first removing cycles and parallel edges, then filling containers, and finally merging containers based on edges. The overall complexity is dominated by the nested operations involving container merging, leading to a quintic complexity.

Input: None

```
containers = new empty list
tree = new empty graph

removeCycles() // O(N2)
removeParallelEdges() // O(N3)
createAndFillContainers(containers) // O(N2)

For each edge in graph do // O(N)
        containerX =
        getContainerWithVertex(edge.getPointX(),
        containers) // O(N2)

        containerY =
        getContainerWithVertex(edge.getPointY(),
        containers) // O(N2)

If containerX != containerY then
        tree.addEdge(edge)
        containerX.addAll(containerY) // O(N)
        containers.remove(containerY) // O(N)

End If
End For
Return tree
End Algorithm
```

Worst-case Time Complexity: $O(N^3)$, where N is the number of edges in the graph.

Conclusion

The time complexity of the treeBuilding algorithm is $O(N^3)$. This is due to the `removeParallelEdges()` function, which can have a worst-case complexity of $O(N^3)$. The other operations in the algorithm have lower complexity, but they are dominated by the `removeParallelEdges()` function.

	A	B
1	Operation	Complexity
2	<code>containers = new empty list</code>	$O(1)$
3	<code>tree = new empty graph</code>	$O(1)$
4	<code>removeCycles()</code>	$O(N^2)$
5	<code>removeParallelEdges()</code>	$O(N^3)$
6	<code>createAndFillContainers(containers)</code>	$O(N^2)$
7	For each edge in graph do	$O(N)$
8	<code>getContainerWithVertex(edge.getPointX(), containers)</code>	$O(N)$
9	<code>getContainerWithVertex(edge.getPointY(), containers)</code>	$O(N)$
10	If <code>containerX != containerY</code> then	$O(1)$
11	<code>tree.addEdge(edge)</code>	$O(1)$
12	<code>containerX.addAll(containerY)</code>	$O(N)$
13	<code>containers.remove(containerY)</code>	$O(1)$

The `removeParallelEdges()` function is the most expensive operation in the algorithm. This is because it has to check every pair of edges in the graph to see if they are parallel. In the worst case, this will take $O(N^3)$ operations.

US 17

- **reconstructPaths ()**

Explanation

The algorithm, `reconstructPaths`, takes an array of predecessors (`prev`), a source vertex (`src`), and the total number of vertices (`n`). It constructs paths from the source vertex to all other vertices based on the provided predecessors. The resulting paths are stored in a list of lists called `paths`. If a vertex has a valid predecessor, the algorithm backtracks through the predecessors to create the path. Finally, it returns the list of paths.

Input: prev (array of predecessors), src (source vertex), n (number of vertices)

paths = new empty list of lists

For i from 0 to n - 1 do // O(N)

path = new empty list

If i != src and prev[i] != -1 then

at = i

While at != -1 do // O(N)

Add at to the beginning of path // O(N)

at = prev[at]

End While

End If

Add path to paths

End For

Return paths

End Algorithm

Worst-case Time Complexity: $O(N^2)$, where N is the number of edges in the graph.

- **Dijkstra()**

Input: graph (adjacency matrix), src (source vertex)

n = length of graph

dist = new array of size n, filled with ∞ // $O(N)$

visited = new array of size n, filled with false // $O(N)$

dist[src] = 0

For i from 0 to n - 2 do // $O(N)$

u = -1

minDist = ∞

For j from 0 to n - 1 do // $O(N)$

If not visited[j] and dist[j] < minDist then

minDist = dist[j]

u = j

End If

End For

If u == -1 then break

visited[u] = true

For v from 0 to n do // $O(N)$

If graph[u][v] > 0 and not visited[v] and dist[u] != ∞ and dist[u] + graph[u][v] < dist[v] then

dist[v] = dist[u] + graph[u][v]

End If

End For

End For

Return dist

End Algorithm

$$O(N) \times (O(N) + O(N)) = O(N) \times O(N) = O(N^2)$$

Worst-case Time Complexity: $O(N^2)$, where N is the number of edges in the graph.

- **DijkstraWithPaths()**

```
Input: graph (adjacency matrix), src (source vertex)

n = length of graph

dist = new array of size n, filled with  $\infty$  //  $O(N)$ 

visited = new array of size n, filled with false //  $O(N)$ 

prev = new array of size n, filled with -1 //  $O(N)$ 

dist[src] = 0

For i from 0 to n - 1 do //  $O(N)$ 

    u = -1

    minDist =  $\infty$ 

    For j from 0 to n do //  $O(N)$ 

        If not visited[j] and dist[j] < minDist then

            minDist = dist[j]

            u = j

        End If

    End For

    If u == -1 then break

    visited[u] = true

    For v from 0 to n do //  $O(N)$ 

        If graph[u][v] > 0 and not visited[v] and dist[u] !=  $\infty$  and dist[u] + graph[u][v] < dist[v] then

            dist[v] = dist[u] + graph[u][v]

            prev[v] = u

        End If

    End For

    End For

    Return reconstructPaths(prev, src, n) //  $O(N^2)$ 

End Algorithm
```

Worst-case Time Complexity: $O(N^2)$, where N is the number of edges in the graph.

Conclusion

Dijkstra's Algorithm:

- The time complexity of the original Dijkstra's algorithm is indeed $O(N^2)$. This is because there are two nested loops, each iterating over the N vertices in the graph. The outer loop iterates at most $N-1$ times (as stated in the analysis). Within the outer loop, the inner loop iterates through all N vertices to find the unvisited vertex with the minimum distance.

Dijkstra's Algorithm with Path Reconstruction:

- The analysis correctly identifies that the core Dijkstra's algorithm portion maintains the same $O(N^2)$ complexity. However, the additional reconstructPaths function introduces a new complexity. The reconstructPaths function iterates through all N vertices and potentially performs another loop to traverse the predecessor array to build individual paths. In the worst case, this nested loop structure can also result in $O(N^2)$ complexity.

Therefore, the total complexity of Dijkstra's algorithm with path reconstruction becomes $O(N^2) + O(N^2) = O(N^2)$. In Big O notation, we consider the dominant term, which in this case is $O(N^2)$.

US 18

The analysis of the time complexities for US 17 is directly applicable to the US18 algorithm as well.

Since US18 utilizes the same core Dijkstra's algorithm for finding the shortest paths and also incorporates the path reconstruction functionality, its time complexity remains the same as the analyzed versions:

- **Dijkstra's Algorithm:** $O(N^2)$
- **Dijkstra's Algorithm with Path Reconstruction:** $O(N^2)$
- **Reconstruct Paths Algorithm:** $O(N^2)$