INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MDISC REPORT

GROUP 132

EDUARDO CARREIRO – 1211199

JOÃO SILVA – 1221293

MARTIM PENEDONES - 1211369

PORTO, 09 DE JUNHO DE 2024

ÍNDEX

1 Introduction

1.1 US13: Infrastructure Optimization

Objective: Apply an algorithm to determine the routes and pipes needed to be laid in a park, minimizing the total cost while ensuring adequate supply to all points.

Constraints:
Use primitive operations only.
Output the subgraph with the routes and edge costs in a .csv file.
Provide visualizations of the input and output graphs.

Work Delivery Format: A folder containing graph visualizations, a .csv file with the output graph and costs, and a .pdf print of the implemented procedures.

1.2 US17: Evacuation Signs to Assembly Point

Objective: Develop an algorithm to place evacuation signs in a park that guide users to an Assembly Point using the shortest route.

Data Input: A .csv file containing a weighted matrix representing the cost between points, and a file listing the points with Assembly Points identified.

Constraints:
The algorithm should use primitive operations only.
Output the shortest route from any sign to the Assembly Point in a .csv file.
Provide visualizations of the input graph and the shortest routes.

Work Delivery Format: A folder containing graph visualizations, a .csv file with the output paths and costs, and a .pdf print of the implemented procedures

1.3 US18: Evacuation Signs to Multiple Assembly Points

Objective: Extend the algorithm to guide park users to the closest Assembly Point using the shortest route.

Data Input: Similar to US17, with multiple Assembly Points.

Constraints and Deliverables: Similar to US17, with additional considerations for multiple destinations.

2 Theoretical Framework

The algorithms for these user stories are based on classical graph theory and optimization techniques. The fundamental concepts include:

- **Shortest Path Algorithms:** Techniques like Dijkstra's algorithms are used to find the shortest paths in weighted graphs. These algorithms are analyzed in terms of their time complexity and suitability for the given problems.
- **Minimum Spanning Tree (MST):** Algorithms such as Kruskal's are employed to find the MST, which is crucial for optimizing infrastructure costs.

3 PSEUDOCODE

3.1 Algorithm one – Kruskal

```
procedure Kruskal(G, E, V, real)
    T := empty set
    for each vertex v in V
        make-set(v)
    sort E by weight
    for each edge (u, v) in E
        if find-set(u) ≠ find-set(v)
            add (u, v) to T
            union(u, v)
        end if
    end for
    return T
end procedure
```

3.1.1 WORST – CASE TIME COMPLEXITY ANALYSIS - Kruskal

Initialization:

T := empty set

This is a simple initialization step and takes constant time, O(1).

Creating Sets for Each Vertex:

for each vertex v in V
  make-set(v)

There are |V| vertices, and make-set(v) is called for each vertex.

Assuming make-set(v) operates in constant time, this loop takes O(|V|) time.

Sorting Edges by Weight:

```
sort E by weight
```

Sorting the edges, E, can be done using an efficient sorting algorithm like Merge Sort or Heap Sort, both of which have a time complexity of O(|E| log |E|).

Processing Each Edge:

```
for each edge (u, v) in E
   if find-set(u) ≠ find-set(v)
      add (u, v) to T
      union(u, v)
   end if
end for
```

The for loop runs for each edge, so it iterates |E| times.

find-set operations are performed twice per edge, and each find-set operation has an amortized time complexity of O(α(|V|)), where α is the inverse Ackermann function, which grows very slowly and is nearly constant for practical input sizes.

If the find-set conditions hold, union is called, which also has an amortized time complexity of O(α(|V|)).

Therefore, the total complexity for this block is O(|E| α(|V|)).

Returning the Result:

```
return T
```

This is a constant time operation, O(1).

Total Worst-case Time Complexity

Combining the complexities of all these steps, the total worst-case time complexity for Kruskal's algorithm is:

O(|V|)+O(|E|log⁡|E|)+O(|E|α(|V|))+O(1)O(|V|) + O(|E| \log |E|) + O(|E| \alpha(|V|)) + O(1)O(|V|)+O(|E|log|E|)+O(|E|α(|V|))+O(1)

Given that α(|V|)\alpha(|V|)α(|V|) is a very slowly growing function and can be considered almost constant, the dominating terms are:

O(|E|log⁡|E|)O(|E| \log |E|)O(|E|log|E|)

Thus, the overall worst-case time complexity of the Kruskal's algorithm is:

O(|E|log⁡|E|)O(|E| \log |E|)O(|E|log|E|)

Explanation of the Lines

- **Initialization (T := empty set)**: This sets up an empty set to store the resulting edges of the minimum spanning tree.
- **Creating Sets (for each vertex v in V make-set(v))**: Initializes each vertex as a separate set, preparing for the union-find operations.
- **Sorting Edges (sort E by weight)**: Sorts all edges based on their weights to process them in ascending order, which is crucial for Kruskal's algorithm.
- **Processing Each Edge (for each edge (u, v) in E)**: Iterates through all edges to determine if they should be included in the minimum spanning tree.
    - **Finding Sets (if find-set(u) ≠ find-set(v))**: Checks if the current edge connects two different sets (i.e., it doesn't form a cycle).
    - **Adding Edge (add (u, v) to T)**: Adds the edge to the result set if it connects two different sets.
    - **Union Sets (union(u, v))**: Merges the sets containing the endpoints of the current edge.
- **Returning the Result (return T)**: Returns the set of edges that form the minimum spanning tree.

3.2 Algorithm two - Dijkstra

```
procedure Dijkstra(G, source, real)
    dist[source] := 0
    for each vertex v in G
        if v ≠ source
            dist[v] := infinity
        end if
        add v to Q
    end for
    while Q is not empty
        u := vertex in Q with min dist[u]
        remove u from Q
        for each neighbor v of u
            alt := dist[u] + length(u, v)
            if alt < dist[v]
                dist[v] := alt
                prev[v] := u
            end if
        end for
    end while
    return dist, prev
end procedure
```

4.1.1 WORST – CASE TIME COMPLEXITY ANALYSIS - Dijkstra

Dijkstra

Line 1: procedure Dijkstra(G, source, real)

- This line is a procedure declaration and does not contribute to time complexity.
- Complexity: *O(1)O(1)*O(1).

Line 2: dist[source] := 0

- Initializes the distance to the source to 0.
- Complexity: *O(1)O(1)*O(1).

Lines 3-7: for each vertex v in G

- A loop that iterates over each vertex in the graph and initializes the distance to infinity (except the source).
- Complexity of the loop: *O(|V|)O(|V|)*O(|V|), where |V| is the number of vertices.
- Initializing dist[v] := infinity and add v to Q are constant time operations *O(1)O(1)*O(1).

Lines 8-18: while Q is not empty

- This main loop runs until the priority queue (Q) is empty.
- Each iteration removes the vertex u with the smallest distance.

Lines 9-10: u := vertex in Q with min dist[u] and remove u from Q

- Extract-min operations.
- Using a binary heap, each extract-min operation takes $O(\log |V|)$ $O(\log |V|)$O(log|V|).
- The while loop runs once for each vertex, resulting in $O(|V|\log |V|)$ $O(|V| \log |V|)$O(|V|log|V|) for all extract-min operations.

Lines 11-16: for each neighbor v of u

- A loop that iterates over each neighbor v of u.
- The complexity is proportional to the number of edges in the graph, resulting in $O(|E|)$ $O(|E|)$O(|E|), where |E| is the number of edges.

Lines 12-15: Updates and comparisons

- Calculating alt := dist[u] + length(u, v) and comparing alt < dist[v] are constant time operations $O(1)$ $O(1)$O(1).
- Updating dist[v] := alt and prev[v] := u also takes $O(1)$ $O(1)$O(1).
- The decrease_priority(v, alt) operation to update the priority in the priority queue takes $O(\log |V|)$ $O(\log |V|)$O(log|V|).
- Since each edge is relaxed once, this results in a total of $O(|E|\log |V|)$ $O(|E| \log |V|)$O(|E|log|V|).

Line 17: return dist, prev

- Returns the distance and predecessor arrays.
- Complexity: $O(1)$ $O(1)$O(1).

Conclusion

Summing up all the time complexities of the individual lines, we get:

- $O(1)$ $O(1)$O(1) (procedure declaration)
- $O(1)$ $O(1)$O(1) (initializing the source)
- $O(|V|)$ $O(|V|)$O(|V|) (initializing all distances)
- $O(|V|\log |V|)$ $O(|V| \log |V|)$O(|V|log|V|) (extract-min operations)
- $O(|E|\log |V|)$ $O(|E| \log |V|)$O(|E|log|V|) (edge relaxations)

Therefore, the total worst-case time complexity of Dijkstra's algorithm is:

$O(|V|)+O(|V|\log |V|)+O(|E|\log |V|)=O((|V|+|E|)\log |V|)$ $O(|V|) + O(|V| \log |V|) + O(|E|\log |V|)=O((|V|E|)\log |V|)$O(|V|)+O(|V|log|V|)+O(|E|log|V|)= O((|V|+|E|)log|V|)

For a connected graph, where $|E|$ $|E|$|E| is at least $|V|-1$ $|V|$ - 1|V|−1 and at most $|V|2$ $|V|^2$|V|2, this can be simplified to:

$O(|E|\log |V|)$ $O(|E| \log |V|)$O(|E|log|V|)

4 Conclusion

This report has comprehensively addressed the user stories US17, US18, and US13, focusing on the development of algorithms for evacuation route optimization and infrastructure cost minimization in a park environment. The chosen algorithms, Dijkstra's for shortest path problems (US17 and US18) and Kruskal's for minimum spanning tree problems (US13), were implemented with a clear focus on efficiency and adherence to primitive operations constraints.

For **US17**, we implemented Dijkstra's algorithm to determine the shortest evacuation routes from various points to a single Assembly Point. The pseudo-code was detailed, and a thorough worst-case time complexity analysis was conducted, confirming its efficiency. The output, including the shortest paths and their associated costs, was provided in a .csv file. Additionally, visualizations of the input graph and the evacuation routes were generated to aid in practical application and verification.

US18 extended the challenge to multiple Assembly Points. The algorithm was adapted to find the nearest Assembly Point for each evacuation sign using Dijkstra's algorithm. This required additional logic to handle multiple destinations efficiently. The results were similarly outputted in a .csv file and visualized, demonstrating the algorithm's effectiveness in a more complex scenario.

For **US13**, we employed Kruskal's algorithm to optimize the park's infrastructure by determining the routes and pipes needed to minimize the total cost while ensuring all points are adequately supplied. The pseudo-code was presented, followed by a detailed analysis of its worst-case time complexity, showcasing its suitability for the task. The output included the subgraph of selected routes and the total cost, presented in a .csv file. Visualizations of both the input and output graphs were created to provide a clear representation of the optimized infrastructure.

The theoretical framework outlined in this report provided a solid foundation for understanding the operations and complexities of the implemented algorithms. By leveraging key concepts from graph theory and optimization, the solutions are not only theoretically sound but also practically efficient, ensuring scalability to larger datasets if necessary.

In conclusion, this report successfully meets the specified requirements of the user stories, demonstrating the importance of algorithmic efficiency and practical application. The combination of detailed pseudo-code, complexity analysis, and visual outputs ensures that the developed algorithms are both reliable and actionable. These solutions provide a robust framework for emergency evacuation planning and

infrastructure optimization, enhancing the safety and operational efficiency of the park environment.