Analysis of Algorithms II

BLG 336E

Assignment 1 Report

Yusuf Yıldız
yildizyu21@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 24.03.2024

1. Implementation Details

1.1. Pseudo-codes of DFS and BFS Algorithms

1.1.1. Depth-First Search Algorithm

Time Complexity:

- $O(N * M)$ for the while loop where N is the number of rows and M is the number of columns, as each cell in the map may be visited once. N * M implies the whole array so we may use "n" to denote input size, and the complexity becomes $O(n)$.
- $O(1)$ for all other operations within the while loop, such as queue operations and cell checks.

Overall Time Complexity: $O(n)$ considering the worst-case scenario where all cells need to be visited. The detailed pseudo-code of the Depth-First Search algorithm is as follows:

```
DFS(map, row, col, resource):
    Initialize colony_size to 0 //O(1)
    Create a stack called location_stack //O(1)
    Push the starting location (row, col) to location_stack //O(1)
    while location_stack is not empty: //O(n) if it traverses all the map
        cur_location ← top element of location_stack //O(1)
        Pop the top element from location_stack //O(1)
        cur_row ← row component of cur_location //O(1)
        cur_col ← column component of cur_location //O(1)
        if visited[cur_row][cur_col] is true or map[cur_row][cur_col] is not equal to resource: //O(1)
            Continue to the next iteration //O(1)
        Mark visited[cur_row][cur_col] as true //O(1)
        Increment colony_size by 1 //O(1)
        for each direction in directions: //O(1) since there is 4 directions, it is constant
            Calculate new_loc_first as (cur_row + direction.first + map.size()) % map.size() //O(1)
            Calculate new_loc_second as (cur_col + direction.second + map[0].size()) % map[0].size() //O(1)
            if map[new_loc_first][new_loc_second] is equal to resource and visited[new_loc_first][new_loc_second]
is false: //O(1)
                Push (new_loc_first, new_loc_second) to location_stack //O(1)

    Return colony_size //O(1)
```

1.1.2. Breadth-First Search Algorithm

Time Complexity:

- $O(N * M)$ for the while loop where N is the number of rows and M is the number of columns, as each cell in the map may be visited once. N * M implies the whole array so we may use "n" to denote input size, and the complexity becomes $O(n)$.
- $O(1)$ for all other operations within the while loop, such as queue operations and cell checks.

Overall Time Complexity: $O(n)$ considering the worst-case scenario where all cells need to be visited. The detailed pseudo-code of the Breadth-First Search algorithm is as follows:

```
BFS(map, row, col, resource):
    Initialize colony_size to 0 //O(1)
    Create a queue called location_queue //O(1)
    Push the starting location (row, col) to location_queue //O(1)
    while location_queue is not empty: //O(n) if it traverses all the map
        cur_location ← front element of location_queue //O(1)
        Pop the front element from location_queue //O(1)
        cur_row ← row component of cur_location //O(1)
        cur_col ← column component of cur_location //O(1)
        if visited[cur_row][cur_col] is true or map[cur_row][cur_col] is not equal to resource: //O(1)
            Continue to the next iteration //O(1)
        Mark visited[cur_row][cur_col] as true //O(1)
        Increment colony_size by 1 //O(1)
        for each direction in directions: //O(1) since there is 4 directions, it is constant
            Calculate new_loc_first as (cur_row + direction.first + map.size()) % map.size() //O(1)
            Calculate new_loc_second as (cur_col + direction.second + map[0].size()) % map[0].size() //O(1)
            if map[new_loc_first][new_loc_second] is equal to resource and visited[new_loc_first][new_loc_second]
is false: //O(1)
                Push (new_loc_first, new_loc_second) to location_queue //O(1)

    Return colony_size //O(1)
```

1.1.3. Top K-Largest Colonies Algorithm

Time Complexity:

- $O(N \times M \times R)$ for the nested loops where N is the number of rows, M is the number of columns, and R is the maximum possible resource type. R is limited between 1 to 9, or it is relatively small to the input N x M, as a result, it can be considered as a constant. N x M implies the whole array so we may use "n" to denote input size, and the complexity becomes $O(n)$.
- For each nested loop, the"DFS" or "BFS" function is called and the time complexities of these functions are given as $O(n)$ above. Thus, the overall complexity of nested loops is $O(n^2)$.
- $O(R \times \log(R))$ for sorting the colonies. This complexity may be negligible among the nested loop's complexity.
- $O(1)$ for all other operations within the function.

Overall Time Complexity: $O(N \times M \times (N \times M \times R) + R \times \log(R))$, or simply $O(n^2)$, considering the worst-case scenarios. The detailed pseudo-code of the top k-largest colonies function is as follows:

```
top_k_largest_colonies(map, useDFS, k):
    Initialize start to the current time //O(1)
    Initialize emptyFlag to true //O(1)
    Initialize topColonies as an empty vector //O(1)
    if map is not empty and map[0] is not empty: //O(1)
        Set emptyFlag to false //O(1)
        Initialize colonies as an empty vector //O(1)
        Initialize visited as a 2D vector of size map with all elements false //O(N × M) where N is the number of
rows and M is the number of columns, simply O(n)
        for resource from 1 to infinity (until containsValue(map, resource) returns false): //O(R × N × M) where R
is the maximum possible resource type, simply O(n) because R can be treated as constant
            for i from 0 to map.size() - 1: //O(N)
                for j from 0 to map[0].size() - 1: //O(M)
                    if useDFS is true: //O(1)
                        Call dfs(map, i, j, resource) and store the result in colonySize //O(N × M), simply O(n)
                    else: //O(1)
                        Call bfs(map, i, j, resource) and store the result in colonySize //O(N × M), simply O(n)
                    if colonySize > 0: //O(1)
                        Append (colonySize, resource) to colonies //O(1)
        Sort colonies in descending order based on colony size //O(R × log(R))
        Take the first min(k, colonies.size()) elements from colonies and assign them to topColonies //O(k)
    Initialize stop to the current time //O(1)
    Calculate the duration as stop - start //O(1)
    Print "Time taken: duration nanoseconds" //O(1)
    if emptyFlag is true: //O(1)
        Return an empty vector //O(1)
    else: //O(1)
        Return topColonies //O(1)
```

1.1.4. Conclusion

In conclusion, we have presented pseudo-codes for the Depth-First Search (DFS) and Breadth-First Search (BFS) algorithms, along with the top k-largest colonies algorithm. Both DFS and BFS have a time complexity of $O(n)$, where n denotes the input size, considering the worst-case scenario. These algorithms traverse the entire map, visiting each cell at most once. Additionally, we provided the pseudo-code for the top k-largest colonies algorithm, which utilizes either DFS or BFS based on user preference. Its time complexity is $O(n^2)$, primarily due to nested loops iterating over the map, with an additional overhead of sorting the colonies in descending order. Overall, these pseudo-codes offer clear insights into the operation of these algorithms and their respective time complexities, facilitating their implementation and understanding.

2. Effects of Maintaining a List of Discovered Nodes

Maintaining a list of discovered nodes is crucial in graph traversal algorithms like Depth-First Search (DFS) and Breadth-First Search (BFS). These algorithms rely on tracking visited nodes to ensure that each node is processed only once, preventing infinite loops and ensuring correctness. The list of discovered nodes, often implemented as a boolean matrix or vector, keeps track of which nodes have been visited during the traversal process.

In DFS, maintaining a list of discovered nodes helps in efficiently exploring the graph or grid by systematically traversing each reachable node. Without this list, DFS might revisit already explored nodes, leading to an inefficient or incorrect traversal. By marking visited nodes, DFS avoids redundant exploration, resulting in a more optimal traversal.

Similarly, in BFS, maintaining a list of discovered nodes ensures that the algorithm explores the graph layer by layer, starting from the source node. Without this list, BFS might revisit nodes at different depths, deviating from the intended layer-by-layer exploration strategy. By marking visited nodes, BFS guarantees that each layer is explored completely before moving to the next layer, ensuring the shortest path is found in unweighted graphs.

Overall, maintaining a list of discovered nodes enhances the efficiency and correctness of graph traversal algorithms like DFS and BFS by preventing redundant exploration and ensuring each node is visited exactly once during the traversal process.

However, there is a trade-off to consider. While maintaining a separate list of discovered nodes incurs a space complexity of $O(n)$, another option is to mark visited nodes directly within the map itself by setting their values to -1. This approach marginally decreases the space complexity but compromises the integrity of the map data. The decision between these options depends on the design considerations of the program and must be made wisely to balance space efficiency with data integrity.

3. Effects of the Map Size on the Performance

The table below is prepared to compare both algorithms' performances in different input data. The time taken is given in nanoseconds(ns).

	Map1	**Map2**	**Map3**	**Map4**
DFS	106.178	15.579	7.864	555.228
BFS	105.056	15.408	7.674	552.603

Table 3.1: Running Time Comparison of DFS and BFS on input data.

3.1. Effects on Time Complexity Performance

- **DFS:**The time complexity of the Depth-First Search (DFS) algorithm is $O(n)$, where n represents the total number of cells in the map. As the map size increases, the number of cells (n) increases, leading to a linear increase in the time complexity. DFS traverses the entire depth of each branch before backtracking, making its time complexity dependent only on the number of cells in the map.

- **BFS:**The time complexity of the Breadth-First Search (BFS) algorithm is also $O(n)$. However, BFS tends to be more sensitive to the width of the graph or grid. In narrower maps, BFS might perform better due to its layer-by-layer traversal, while in larger maps, the performance might degrade due to the increased number of nodes at each layer. This sensitivity to width arises from BFS's exploration strategy, where it visits all neighbors of a node before moving on to their children.

3.2. Effects on Space Complexity Performance

Both DFS and BFS algorithms typically require space to store the visited nodes or cells. The space complexity for both algorithms is also $O(n)$, where n represents the total number of cells in the map. As the map size increases, the space required to store the visited nodes also increases linearly. This is because the size of the visited array (or matrix) grows with the number of cells in the map. Additionally, both DFS and BFS utilize additional space for their traversal stacks or queues, contributing to the linear space complexity.

3.3. Conclusion

Overall, increasing the map size leads to a linear increase in both time and space complexity for both DFS and BFS algorithms. However, BFS might exhibit slightly different performance characteristics compared to DFS, particularly in narrower or larger maps, due to its layer-by-layer traversal strategy.

4. Effects of the Algorithm Type (DFS & BFS) on the Performance

From the provided data given in the previous chapter, it can be concluded that:

1. **Map1**: This map is in size of 8×8. The running times for both DFS and BFS are quite close, with BFS slightly outperforming DFS by a small margin.
2. **Map2**: This map is in size of 2×5. Similarly, both algorithms exhibit similar running times, with BFS being marginally faster than DFS.
3. **Map3**: This map is in size of 1×2. Once again, DFS and BFS demonstrate comparable performance, with BFS being slightly faster than DFS.
4. **Map4**: This map is in size of 23×16. In this case, both DFS and BFS show significant differences in running times, with BFS performing slightly better than DFS.

Overall, the choice between DFS and BFS appears to have a minimal impact on the performance of finding the largest colony in these specific scenarios. Both algorithms generally exhibit similar running times across different map sizes, with occasional variations depending on the specific characteristics of the input data. However, in narrower or larger maps (such as Map4), BFS might exhibit slightly better performance compared to DFS, as it can efficiently explore layers of nodes in a breadth-first manner.

5. Conclusion

In this report, we have provided a comprehensive analysis of graph traversal algorithms, focusing on Depth-First Search (DFS) and Breadth-First Search (BFS), along with their application in finding the largest colony in a given map.

First, we presented detailed pseudo-codes for DFS and BFS algorithms, outlining their operation and time complexities. Both algorithms exhibit a time complexity of $O(n)$, where n represents the total number of cells in the map. Additionally, we introduced the top k-largest colonies algorithm, which utilizes either DFS or BFS based on user preference, with a time complexity of $O(n^2)$.

Next, we discussed the importance of maintaining a list of discovered nodes in graph traversal algorithms. By tracking visited nodes, these algorithms ensure correctness and prevent redundant exploration, thereby enhancing efficiency. However, there is a trade-off between maintaining a separate list and directly marking visited nodes within the map, which must be carefully considered based on space efficiency and data integrity requirements.

Furthermore, we examined the effects of map size on the performance of DFS and BFS algorithms. Both time and space complexities increase linearly with map size, with BFS being more sensitive to the width of the graph due to its layer-by-layer traversal strategy.

Finally, we investigated how the choice between DFS and BFS affects the performance of finding the largest colony. While both algorithms generally exhibit similar running times, BFS may outperform DFS in narrower or larger maps due to its traversal strategy.

Overall, this report highlights the importance of understanding the intricacies of graph traversal algorithms and considering various factors such as map size and traversal strategy to optimize performance in real-world applications.