Analysis of Algorithms 2

BLG 336E

Project 1 Report

Abdullah Jafar Mansour Shamout
shamout21@itu.edu.tr

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

1. Graph Algorithms

1.1. Depth First Search(DFS)

1.1.1. Pseudo-code

```
procedure DFS(map, row, col, resource)
    size = 0
    rows = size of map
    cols = size of map[0]
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  // defining the directions that I can go in the map

    dfs_stack = empty stack  // defining a stack for the implementation of the dfs algorithm
    push (row, col) to dfs_stack  // starting the process of dfs by pushing a point on the map to the stack

    while dfs_stack is not empty do  // while the stack is not empty
        current = top element of dfs_stack  // get the value at the top of the stack
        pop element from dfs_stack  // pop it from the stack
        row = first element of current  // get its row
        col = second element of current  // get its column

        if (row >= 0 and row < rows and col >= 0 and col < cols and map[row][col] == resource) then  // check if that point is valid and is not visited before
            map[row][col] = -1  // Mark current cell as visited by changing its resource value to -1
            size = size + 1  // Increment colony size

            // Explore all four directions
            for each dir in directions do
                new_r = (row + dir.first + rows) modulo rows  // Handle circularity
                new_c = (col + dir.second + cols) modulo cols  // Handle circularity
                push (new_r, new_c) to dfs_stack  // Push the new point to the stack
            end for
        end if
    end while

    return size
end procedure
```

Listing 1.1: DFS Pseudocode

The pseudocode essentially represents a Depth-First Search (DFS) algorithm tailored for a 2D map structure. By starting from a designated point on the map, indicated by the provided row and column coordinates. As we traverse, we check neighboring cells in four directions: up, down, left, and right. Each time we visit a cell, we mark it as explored by changing its resource value to -1. To keep track of where we've been and where to go next, we use a stack. This allows us to maintain the depth-first traversal behavior. We continue this exploration until our stack is empty, incrementing the size counter for each valid cell encountered. Additionally, to handle boundary conditions when exploring adjacent cells, we've implemented circular indexing. This ensures that we wrap around to the opposite side of the map if we hit the edges.

1.1.2. Time complexity

Analyzing the time complexity of the Depth-First Search (DFS) algorithm presented in the pseudocode, we can observe that the algorithm visits each cell of the 2D map exactly once. In the worst-case scenario, where all cells need to be explored, the algorithm will traverse through every cell of the map. Let n be the number of rows and m be the number of columns in the map. Therefore, the time complexity of the DFS algorithm is $O(n \times m)$, as it linearly depends on the dimensions of the map. Additionally, since the algorithm employs a stack to keep track of visited cells, the space complexity is also $O(n \times m)$ in the worst case, considering the stack may store information for each cell.

1.2. Breadth First Search (BFS)

1.2.1. Pseudo-code

```
procedure BFS(map, row, col, resource)
    size = 0
    rows = size of map
    cols = size of map[0]
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  // defining the directions
   that I can go in the map

    bfs_queue = empty queue  // defining a queue for the implementation of the
   bfs algorithm
    push (row, col) to bfs_queue  // starting the process of bfs by pushing a
   point on the map to the queue
```

```

    while bfs_queue is not empty do  // while the queue is not empty
        current = front element of bfs_queue  // get the value at the front of the queue
        pop element from bfs_queue  // pop it from the queue
        row = first element of current  // get its row
        col = second element of current  // get its column

        if (row >= 0 and row < rows and col >= 0 and col < cols and map[row][col] == resource) then  // check if that point is valid and is not visited before
            map[row][col] = -1  // Mark current cell as visited by changing its resource value to -1
            size = size + 1  // Increment colony size

            for each dir in directions do  // Explore all four directions
                new_r = (row + dir.first + rows) modulo rows  // Handle circularity
                new_c = (col + dir.second + cols) modulo cols  // Handle circularity
                push (new_r, new_c) to bfs_queue
            end for
        end if
    end while

    return size
end procedure
```

Listing 1.2: BFS Pseudocode

1.2.2. Time complexity

Analyzing the time complexity of the Breadth-First Search (BFS) algorithm presented in the pseudocode, similar to the DFS algorithm, we can observe that the algorithm visits each cell of the 2D map exactly once. However, unlike DFS, BFS prioritizes exploring cells in a breadth-wise manner, ensuring that cells are visited in order of their distance from the starting point. As a result, BFS traverses through every cell of the map in a level-by-level manner. Let n be the number of rows and m be the number of columns in the map. Therefore, the time complexity of the BFS algorithm is also $O(n \times m)$. Additionally, the space complexity of BFS is $O(n \times m)$ in the worst case, as it also employs a queue to keep track of visited cells, potentially storing information for each cell.

1.3. Top-K Largest Colonies

1.3.1. Pseudo-code

```
procedure top_k_largest_colonies(map, useDFS, k)
    start = high_resolution_clock::now()  // Start measuring time

    if (map is empty) then  // if the map is empty
        return {}  // return an empty vector
    end if

    rows = size of map
    cols = size of map[0]
    result = empty vector  // to store the result of the top-k largest colonies

    for i = 0 to rows - 1 do  // loop over the rows of the map
        for j = 0 to cols - 1 do  // loop over the columns of the map
            if (map[i][j] is not -1) then  // if the cell is not visited before
                resource = map[i][j]  // get the resource value of the cell
                if (useDFS is true) then
                    size = dfs(map, i, j, resource)  // get the size of the colony using DFS
                else
                    size = bfs(map, i, j, resource)  // get the size of the colony using BFS
                end if
                result.push_back({size, resource})  // push the size and the resource value of the colony to the result vector
            end if
        end for
    end for

    stop = high_resolution_clock::now()  // Stop measuring time
    duration = duration_cast<nanoseconds>(stop - start)  // Calculate the duration
    print "Time taken: " + duration + " nanoseconds"  // Print the time taken

    // sort the result by size in descending order; if sizes are equal, then sort them by resource type in ascending order
    sort(result.begin(), result.end(), [](pair<int, int> x, pair<int, int> y)
         {
        if (x.first is equal to y.first) then
            return x.second < y.second
        end if
        return x.first > y.first })

    if (size of result is greater than k) then
        resize result to size k  // Keep only the top-k largest colonies
    end if

    return result
end procedure
```

Listing 1.3: Top-K Largest Colonies Pseudocode

1.3.2. Time complexity

The time complexity of the ‘top_k_largest_colonies‘ function theoretically depends on the choice between using Depth-First Search (DFS) or Breadth-First Search (BFS) algorithms. Both DFS and BFS algorithms have a time complexity of $O(n \times m)$, where n is the number of rows and m is the number of columns in the map. Since the function iterates through each cell of the map and calls either DFS or BFS for each unvisited cell, the overall time complexity remains $O(n \times m)$. Additionally, sorting the result by colony size adds a negligible overhead, so the time complexity of sorting can be considered as $O(k \log k)$, where k is the number of colonies in the result vector. Therefore, the overall time complexity of the ‘top_k_largest_colonies‘ function is $O(n \times m + k \log k)$.

1.4. Why should you maintain a list of discovered nodes? How does this affects the outcome of the algorithms?

Maintaining a list of discovered nodes is crucial in graph traversal algorithms like Depth-First Search (DFS) and Breadth-First Search (BFS). This list helps in ensuring that each node is visited only once during the traversal process. Without maintaining this list, algorithms might revisit already visited nodes, infinite loops in certain cases.

In DFS, without keeping track of visited nodes, the algorithm might get stuck in cycles indefinitely, as it keeps revisiting the same nodes. This would result in incorrect outputs and potentially infinite runtime. Similarly, in BFS, not maintaining a list of discovered nodes could lead to redundant exploration of nodes, affecting the efficiency of the algorithm.

Maintaining a list of discovered nodes ensures the correctness and efficiency of graph traversal algorithms by preventing revisits to already explored nodes and enabling the algorithms to terminate appropriately.

1.5. How does the map size affect the performance of the algorithm for finding the largest colony in terms of time and space complexity?

The size of the map directly influences the performance of algorithms for finding the largest colony in terms of both time and space complexity. As the size of the map increases, the number of cells to explore grows, leading to longer execution times and increased memory usage.

In terms of time complexity, both Depth-First Search (DFS) and Breadth-First Search (BFS) algorithms have a time complexity of $O(n \times m)$, where n is the number of rows and m is the number of columns in the map. This means that the time taken by the algorithm increases linearly with the size of the map. Larger maps require traversing more cells, resulting in longer execution times.

Regarding space complexity, both algorithms also have a space complexity of $O(n \times m)$ in the worst case. This is because they typically utilize data structures such as stacks (for DFS) or queues (for BFS) to keep track of visited nodes. As the map size increases, the amount of memory required to store information about visited nodes also increases proportionally.

In summary, larger map sizes result in longer execution times and increased memory usage for algorithms for finding the largest colony, as both time and space complexity scale linearly with the size of the map. Such a result can be seen in the results section of this report. It is demonstrated by MAP 4.

1.6. How does the choice between Depth-First Search (DFS) and Breadth-First Search (BFS) affect the performance of finding the largest colony?

This is discussed and demonstrated in the results section of the report where I do a comparison between the maps with respect to the algorithm used.

1.7. Results

Table 1.1 presents the comparison of execution times for Depth-First Search (DFS) and Breadth-First Search (BFS) algorithms across various maps. Each entry in the table represents the time taken in microseconds (μs) for each algorithm to complete its traversal on the respective map.

	MAP 1	**MAP 2**	**MAP 3**	**MAP 4**	**MAP 5**
DFS	49.2 μs	9.0 μs	7.2 μs	239.5 μs	–
BFS	47.6 μs	9.1 μs	5.7 μs	228.5 μs	–

Table 1.1: Comparison of DFS and BFS times taken per map

As observed from the table, the execution times of both DFS and BFS algorithms vary across different maps. For instance, on MAP 1, DFS took approximately 49.2 microseconds, while BFS took 47.6 microseconds. Similarly, on MAP 2, DFS and BFS took 9.0 and 9.1 microseconds, respectively.

Comparing the overall performance between DFS and BFS, it is evident that BFS tends to be slightly faster than DFS on most maps. This can be observed from the smaller execution times of BFS in comparison to DFS for MAPs 1, 2, and 3.

However, on MAP 4, the execution time of DFS (239.5 microseconds) is noticeably higher compared to BFS (228.5 microseconds). This suggests that the characteristics of the map, such as its size or structure, influences the performance of the algorithms.

Notably, for MAP 5, the execution times for both DFS and BFS are not available (–), indicating that the map is empty and there is no input.

In conclusion, while BFS generally exhibits slightly better performance compared to DFS across the tested maps, the specific characteristics of each map can influence the relative efficiency of the algorithms.