Analysis of Algorithms II

BLG 336E

Project 1 Report

Mustafa Can Çalışkan
caliskanmu20@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering

1. Implementation

1.1. Pseudo-Code

Algorithm 1 Insert Sorted Algorithm

```
function insert_sorted(vec, new_pair)
    if vec is empty then
        vec.push_back(new_pair)
        return
    end if
    for iter $\leftarrow$ vec.begin(); iter $\neq$ vec.end(); iter $\leftarrow$ next element do
        if (new_pair.first $>$ iter.first) then
            vec.insert(iter, new_pair)
            return
        else if (new_pair.first $=$ iter.first and new_pair.second $<$ iter.second) then
            vec.insert(iter, new_pair)
            return
        end if
    end for
    vec.push_back(new_pair)
end function
```

This function adds the found resource-size pairs to a vector in descending order (placing the one with lower resource to the left of the higher one if the same size is present in the vector).

In the worst-case scenario, where the existing elements need to be traversed from beginning to end to insert the new element, the time complexity is $O(n)$.

Algorithm 2 DFS Algorithm

```
1:  function dfs(map, row, col, resource)
2:      total_count ← $0$ |c
3:      row_size ← size of rows in map
4:      col_size ← size of columns in map
5:      limit_row ← $($row $+$ row_size$)$ $\bmod$ row_size
6:      limit_col ← $($col $+$ col_size$)$ $\bmod$ col_size
7:      if map[limit_row][limit_col] $\neq$ resource then
8:          return $0$
9:      end if
10:     map[limit_row][limit_col] ← $-\infty$
11:     total_count ← total_count $+ 1$
12:     total_count ← total_count $+$ dfs(map, limit_row $+ 1$, limit_col, resource)
13:     total_count ← total_count $+$ dfs(map, limit_row $- 1$, limit_col, resource)
14:     total_count ← total_count $+$ dfs(map, limit_row, limit_col $+ 1$, resource)
15:     total_count ← total_count $+$ dfs(map, limit_row, limit_col $- 1$, resource)
16:     return total_count
17: end function
```

This function applies DFS (Depth-First Search) on the map as intended.

The DFS function is recursively called with all neighbors of the location provided as a parameter. The base condition is whether the given location contains the resource or not. Since the map is circular, to prevent boundaries from overflowing, the modulo operation is applied to rows and columns after adding their own sizes to them. The visited positions are marked with -∞ (INT_MIN in C++). Similar to the stack logic, the DFS function is recursively called with the neighbors of the current position.

Since the time complexity is directly dependent on the number of points, and since each node is visited at most once, the total time complexity is $O(mn)$.

Algorithm 3 BFS Algorithm

```
function bfs(map, row, col, resource)
    bfs_queue $\leftarrow$ empty queue
    total_count $\leftarrow 0$
    row_size $\leftarrow$ size of rows in map
    col_size $\leftarrow$ size of columns in map
    bfs_queue.push(make_pair(row, col))
    map$[row][col] \leftarrow -\infty$
    neighbor_directions $\leftarrow \{\{0,1\},\{0,-1\},\{1,0\},\{-1,0\}\}$
    while bfs_queue is not empty do
        (row, col) $\leftarrow$ bfs_queue.front()
        bfs_queue.pop()
        total_count $\leftarrow$ total_count $+ 1$
        for $i \leftarrow 0$ to $3$ do
            neighbor_row $\leftarrow$ neighbor_directions$[i][0]$
            neighbor_col $\leftarrow$ neighbor_directions$[i][1]$
            limit_row $\leftarrow$ (neighbor_row + row + row_size) $\bmod$ row_size
            limit_col $\leftarrow$ (neighbor_col + col + col_size) $\bmod$ col_size
            if map[limit_row][limit_col] = resource then
                map[limit_row][limit_col] $\leftarrow -\infty$
                bfs_queue.push(make_pair(limit_row, limit_col))
            end if
        end for
    end while
    return total_count
end function
```

This function applies BFS (Breadth-First Search) on the map as intended.

Firstly, the given location is added to the queue. Subsequently, until the queue is completely emptied, the neighbors of the first element in the queue are identified. If the neighbors have not been visited before and possess the same resource, they are added to the queue. Then, the first element is removed from the queue.Due to the circular nature of the map, boundary protection is implemented similarly to DFS. The visited positions are marked with $-\infty$ (INT_MIN in C++).

Similar to DFS, since each node will be visited at most once, and the time complexity is directly dependent on the number of nodes, the total time complexity is $O(mn)$.

Algorithm 4 Top-K Largest Colonies Algorithm

```
1: **function** top_k_largest_colonies(map, useDFS, $k$)
2:     region $\leftarrow 0$
3:     result_vector $\leftarrow$ empty vector
4:     **for** $i \leftarrow 0$ **to** size of rows in map $- 1$ **do**
5:         **for** $j \leftarrow 0$ **to** size of columns in map $- 1$ **do**
6:             resource $\leftarrow$ map$[i][j]$
7:             **if** resource $\neq -\infty$ **then**
8:                 **if** useDFS **then**
9:                     region $\leftarrow$ dfs(map, $i, j$, resource)
10:                **else**
11:                    region $\leftarrow$ bfs(map, $i, j$, resource)
12:                **end if**
13:                insert_sorted(result_vector, make_pair(region, resource))
14:            **end if**
15:        **end for**
16:    **end for**
17:    **if** $k <$ size of result_vector **then**
18:        result_vector.resize($k$)
19:    **end if**
20:    **return** result_vector
21: **end function**
```

In this function, all previously unvisited locations are traversed sequentially, applying the desired DFS/BFS operations to the locations, and finally, the resource-size pairs it possesses are added to the result vector in decreasing order while preserving the sequence.

Due to the nested for loops, iterating over each node is $O(mn)$, the number of operations performed for each call to the DFS and BFS functions is constant (worst-case scenario is $O(mn)$, but this can occur only once throughout all iterations), and because the vector is sorted in each loop iteration is $O(x)$ (where x represents the size of the vector), the total time complexity becomes $O(mnx)$.

2. Discussions

1. **Why should we maintain a list of discovered nodes? How does this affects the outcome of the algorithms?**

   During DFS and BFS operations, locations can be visited at most once. If visited locations are not marked, they can be visited again, potentially leading the program into an infinite loop.

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

   Since the time complexity of the ’top_k_largest_colonies’ operation is $O(mnx)$, as the map size increases, the algorithm’s runtime increases proportionally with the map’s width. Since the DFS function calls itself recursively (affecting the call stack directly), the BFS function’s queue continuously expands, and additions are made continuously to the result vector, the space complexity is also significantly influenced by the map’s width.

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

   The fact that both algorithms have a time complexity of $O(mn)$ can be observed in the following table, where they complete in similar durations across different map sizes. However, it has been observed that in the long term (when example files are sorted according to their sizes as much as possible), DFS performs more efficiently compared to BFS. Due to the limited number of tests and the unknown distribution of resources in the files, I do not believe a definitive inference can be made regarding the superior performance of DFS over BFS.

Maps (arranged by size)	Depth-First Search (DFS)	Breadth-First Search (BFS)
Map 5	588972	597703
Map 3	10931227	16814527
Map 2	10931087	16729041
Map 1	14922732	21178357
Map 4	20113695	28294288

Table 2.1: Work durations measured on different maps for k=5 (in nanoseconds)