Analysis of Algorithms II

BLG 336E

Project 1 Report

ERENCULHACI
150220763
culhaci22@itu.edu.tr

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

0.1. Implementation Details

- The Depth First Search function finds a colony on a map starting with a specific resource number in given row and cell.
- The algorithm uses a while loop until the stack s is empty. In each loop itearation, it gets the top element which is the cell index from stack and sets it as visited and increments the size of the colony.
- Next, it examines the adjacent cells for the current cell. If an adjacent cell has the same resource and not been visited before, it's pushed to stack.
- Code checks if row or column index go out of bounds. If they go out of bounds it wraps to opposite side of map. So the map is circular. Finally it returns the size of colony.
- In the worst-case scenario in an m x n map, if every cell contains the resource, the time complexity goes $O(m * n)$.

```
dfs(map, row, col, resource, visited):
    size = 0 // Initialize colony size to 0
    s = empty_stack // Create a stack to store cell indices
    s.push((row, col)) // Push the starting cell to the stack

    while s is not empty: // While the stack is not empty
        p = s.top() // Get the top element from the stack
        s.pop() // Remove the top element from the stack
        r = p.first // Get the row index
        c = p.second // Get the column index

        // Wrap around if indices are out of bounds
        if r < 0: r = number_of_rows(map) - 1
        else if r >= number_of_rows(map): r = 0
        if c < 0: c = number_of_columns(map) - 1
        else if c >= number_of_columns(map): c = 0

        if map[r][c] != resource OR visited[r][c]: // If the cell doesn't contain the resource or is visited
            continue // Skip this cell

        visited[r][c] = true // Mark the cell as visited
        size = size + 1 // Increment the colony size

        // Add neighboring cells to the stack
        s.push((r - 1, c)) // Move north
        s.push((r + 1, c)) // Move south
        s.push((r, c - 1)) // Move west
        s.push((r, c + 1)) // Move east

    return size // Return the colony size
```

Figure 0.1: dfs

- The Breadth First Search function finds a colony on a map starting with a specific resource number in given row and cell. It uses a similar algorithm logic with depth first search but it goes until the queue is emptied.
- In the worst-case scenario in an m x n map just like the DFS, if every cell contains the resource, the time complexity goes $O(m * n)$.

```
bfs(map, row, col, resource, visited):
    size = 0 // Initialize colony size to 0
    q = empty_queue // Create a queue to store cell indices
    q.push((row, col)) // Push the starting cell to the queue

    while q is not empty: // While the queue is not empty
        p = q.front() // Get the front element from the queue
        q.pop() // Remove the front element from the queue
        r = p.first // Get the row index
        c = p.second // Get the column index

        // Wrap around if indices are out of bounds
        if r < 0: r = number_of_rows(map) - 1
        else if r >= number_of_rows(map): r = 0
        if c < 0: c = number_of_columns(map) - 1
        else if c >= number_of_columns(map): c = 0

        if map[r][c] != resource OR visited[r][c]: // If the cell doesn't contain the resource or is visited
            continue // Skip this cell

        visited[r][c] = true // Mark the cell as visited
        size = size + 1 // Increment the colony size

        // Add neighboring cells to the queue
        q.push((r - 1, c)) // Move north
        q.push((r + 1, c)) // Move south
        q.push((r, c - 1)) // Move west
        q.push((r, c + 1)) // Move east

    return size // Return the colony size
```

Figure 0.2: bfs

- Top k largest colonies function finds top k largest colonies and returns them in sorted order.
- First it starts the timer for measuring time.
- Then it checks if the map is empty. And if not, it creates a 2D vector to store visited cells and creates an empty colonies vector to store colony size and resource type pairs.
- Then it iterates using DFS or BFS by deciding with checking useDFS flag.
- Then it uses sort function to sort found colonies in descending order of size and ascending order of resource type.
- Then it creates empty result vector to store top colonies and again sort result by descending order of size and ascending order of resource type and print the results after calculating the duration.
- The dominant factor in the time complexity is the nested loop, so it decides the time complexity which is $O(rows * columns)$.

```
top_k_largest_colonies(map, useDFS, k):
    start_time = current_time() // Start measuring time

    rows = number_of_rows(map) // Get the number of rows
    cols = 0 // Initialize columns count
    if map is not empty:
        cols = number_of_columns(map) // Get the number of columns
    if rows == 0 OR cols == 0: // If the map is empty
        return empty_vector // Return empty vector

    visited = create_2D_vector(rows, cols, false) // Create a 2D vector to store visited cells
    colonies = empty_vector // Vector to store colony size and resource type pairs

    for i from 0 to rows-1: // Iterate over rows
        for j from 0 to cols-1: // Iterate over columns
            if not visited[i][j]: // If cell is not visited
                resource = map[i][j] // Get resource type
                size = 0
                if useDFS:
                    size = depth_first_search(map, i, j, resource, visited) // Use DFS
                else:
                    size = breadth_first_search(map, i, j, resource, visited) // Use BFS
                add (size, resource) to colonies // Add colony size and resource type to colonies vector

    sort colonies by descending order of size and ascending order of resource type // Sort colonies vector

    result = empty_vector // Vector to store top-k largest colonies
    for i from 0 to minimum of k and colonies.size() - 1: // Iterate over colonies vector
        add colonies[i] to result // Add top-k largest colonies to result vector

    sort result by descending order of size and ascending order of resource type // Sort result vector again

    stop_time = current_time() // Stop measuring time
    duration = stop_time - start_time // Calculate duration
    print "Time taken: " + duration + " nanoseconds" // Print time taken

    return result // Return top-k largest colonies
```

Figure 0.3: top k largest colonies

0.2. Questions

1. **Why should you maintain a list of discovered nodes? How does this affects the outcome of the algorithms?** Since we do not want to visit the same node again after visiting once, we keep a list of visited nodes in a visited vector. It prevents infinite loops since the algorithm may visit same nodes infinitely in a loop if a visited vector is not kept. Also it becomes sure that all the nodes are visited once. And by keeping a visited vector algorithm focus on unvisited part of map so improves discovery.

2. **How does the map size affect the performance of the algorithm for finding the largest colony in terms of time and space complexity?** As the time complexity of both BFS and DFS are $O(V + E)$ V(Vertices), E(Edges), as the map size grows bigger, the execute time will grow linearly in both scenarios as it can be seen in the table below.

	Map 1	Map 2	Map 3	Map 4
DFS	35500 ns.	9600 ns.	6700 ns.	174900 ns.
BFS	38100 ns.	9900 ns.	7600 ns.	158600 ns.

Table 1: Execution time on k = 3

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

between DFS and BFS to find largest colony depends on factors such as the map structure and the distribution of colonies. Their affect on performance is nearly same since their complexities are both $O(V + E)$ where V(Vertices) and E(Edges) and it can be seen in the table 1 above. BFS is generally more efficient on maps where the largest colony is closer to the start or distributed evenly across the map since it searchs by exploring level by level whereas DFS can be more efficient on maps with long, narrow paths or when the biggest colony is located deeper in the map since it searches first going into deep inside the map.