Efficient Map Analysis for Connected Galactic Colonies

Exploring Colonies Using DFS and BFS on Circular Maps

BLG 336E - Analysis of Algorithms II
Istanbul Technical University
2024

An algorithmic exploration engine for connected galactic colonies using circular maps. Implements Depth-First Search (DFS) and Breadth-First Search (BFS) to identify and rank top-k largest colonies by resource type. Demonstrates efficient graph traversal and resource mapping in complex spatial networks.

Note: Personal and professor-specific details have been removed to protect privacy.

Explain your code and your solution by

– Writing pseudo-code for your functions following the pseudo-code conventions given in the class slides.

– Show the time complexity of your functions on the pseudo-code.

MY EXTRA FUNCTİONS AT THE BEGİNNİNG

customComparator Function

```
/* START YOUR CODE HERE */
typedef pair<int, int>  pii;

bool    customComparator(const pii  &a, const pii   &b)
{
    if (a.first != b.first)
        return a.first > b.first;
    return a.second < b.second;
}
```

Pseudo Code od customComparator Function

Function customComparator(a: pair<int, int>, b: pair<int, int>) -> bool
If a.first != b.first Then
Return a.first > b.first
Else
Return a.second < b.second
End Function

Time Complexity: O(1) Compares the second elements if the first ones are equal, yet still operates in constant time. Constant time operation as it only involves comparison. Θ(1) Regardless of whether it compares first or both elements, it performs a constant number of steps. Ω(1) - The function returns immediately if the first elements are different, but this does not affect its constant time complexity
This function eases the test process, as it makes the outcomes in order

fix_row_col Function

```
void    fix_row_col(vector<vector<int>> &map, int &row, int &col)
{
    int rows = static_cast<int>(map.size());
    int cols = static_cast<int>(map[0].size());

    row = (row + rows) % rows;
    col = (col + cols) % cols;
}
/* END YOUR CODE HERE */
```

Pseudo Code od fix_row_col Function

Function fix_row_col(map: 2D vector of int, row: int reference, col: int reference) rows = map.size() cols = map[0].size() row = (row + rows) % rows col = (col + cols) % cols End Function
Time Complexity: Worst Case O(1).: Despite any initial values of row and col, the function's execution time does not vary, maintaining constant complexity. Best Case Ω(1): Immediately performs arithmetic operations to adjust row and col, requiring constant time. Average Case: The operation count remains unchanged across all inputs, ensuring a constant time complexity. Θ(1)
The row and col values are the indexes on the map. Thanks to this funtion, in case of exceed the border of the map, indexes return the other side of the map as it requested on the assignment which makes map circular.

DFS Function

```cpp
int dfs(vector<vector<int>>& map, int row, int col, int resource){

    // Initialize size of the colony and push the starting cell onto the stack
    int size = 0;
    stk.push({row, col});

    while (!stk.empty()) {
        pii current = stk.top();
        stk.pop();

        int currentRow = current.first;
        int currentCol = current.second;
        fix_row_col(map, currentRow, currentCol);

        // Check if the current cell is already visited or not of the desired resource
        if (map[currentRow][currentCol] != resource) continue;

        // Mark the cell as visited by setting it to a negative value
        map[currentRow][currentCol] = -1;
        size++; // Increment the size of the colony

        // Push the four neighboring cells onto the stack
        vector<pii> directions = {{0, 1}, {0, -1}, {1, 0}, {-1, 0}};
        for (auto& dir : directions) {
            int newRow = currentRow + dir.first;
            int newCol = currentCol + dir.second;
            fix_row_col(map, newRow, newCol);
            // Check boundaries and if the cell is already visited in fix_row_col function
            if (map[newRow][newCol] == resource) {
                stk.push({newRow, newCol});
            }
        }
    }
    return size; // Return the total size of the colony

    /* END YOUR CODE HERE */
}
```

Pseudo Code of DFS Function

```
DFS_WITH_STACK(map, start_row, start_col, target_resource)
    Create an empty stack called STACK
    Call FIX_ROW_COL to ensure start_row and start_col are within bounds

    IF map[start_row][start_col] is NOT target_resource OR is marked visited THEN
        RETURN 0

    Initialize colony_size to 0
    PUSH (start_row, start_col) onto STACK

    WHILE STACK is not empty DO
        current = POP top element from STACK

        IF map[current.row][current.col] is marked visited THEN
            CONTINUE to the next iteration of the loop

        Mark map[current.row][current.col] as visited
        INCREMENT colony_size by 1
```

```
    FOR EACH direction in [UP, DOWN, LEFT, RIGHT] DO
        next_row = current.row + direction.row_offset
        next_col = current.col + direction.col_offset
        Call FIX_ROW_COL for next_row and next_col

        IF map[next_row][next_col] is target_resource THEN
            PUSH (next_row, next_col) onto STACK

    RETURN colony_size
```

Time Complexity: Time Complexity Analysis:
O(N*M),

Each cell that matches the target resource is visited exactly once due to the marking of visited cells.
For each visited cell, the algorithm performs a constant number of operations (marking the cell and examining up to four neighbors).

In the worst-case scenario, where the DFS explores a large portion of the map, every cell that matches the target resource is pushed to and popped from the stack exactly once.

Conclusion:
The time complexity is O(V), with V being the number of cells visited during the DFS. In a densely populated map, where many cells match the target resource, this can scale up to O(N*M), reflecting the total number of cells in the map. The complexity directly relates to the distribution and connectivity of the target resource cells within the map.

Best Case: The starting cell is not the target resource or already visited, resulting in immediate return. Time Complexity: Ω (1).

Average Case: The function typically explores a portion of the grid before finding all reachable target resource cells, making its complexity proportional to the number of cells it visits. Time Complexity: Θ (N), where N is the number of cells in the grid it actually explores, which can be significantly less than the total number of cells in the grid.

The dfs function initiates a depth-first search from a specified cell (row, col) on a 2D map, targeting cells with a specific resource value. It uses a stack data structure to track the cells to be explored, ensuring a last-in, first-out (LIFO) order of exploration.

Upon starting, the function adjusts the initial row and column indices to ensure they fall within the map's boundaries. It immediately returns zero if the starting cell does not match the desired resource or has already been visited.

For valid starting points, the function proceeds by pushing the starting cell onto the stack. It then enters a loop that continues as long as there are cells to explore in the stack. In each iteration, it pops a cell from the stack, marks it as visited by setting its value to a negative number, and increments a counter that tracks the size of the colony being explored.

The function then examines all four neighboring cells (up, down, left, right) of the current cell. If a neighbor matches the target resource and has not been visited, it is added to the stack for future exploration. This process allows the function to explore the map in a depth-first manner, thoroughly examining each branch before moving on to the next.

By marking cells as visited and only exploring unvisited, resource-matching cells, the function ensures that each cell is counted exactly once. The search terminates when no further cells are left to explore, at which point the function returns the total size of the colony, represented by the number of cells that share the specified resource value and are connected to the starting cell.

This implementation is efficient for exploring complex map structures and identifying contiguous regions (colonies) of interest, utilizing the stack to manage the exploration depth and backtrack as necessary.

BFS Function

```cpp
int bfs(vector<vector<int>>& map, int row, int col, int resource) {
  /* START YOUR CODE HERE */

    queue<pii> q; // Kullanılacak kuyruk veri yapısı
    fix_row_col(map, row, col); // Harita sınırlarını düzelt

    // Eğer başlangıç noktası aranan kaynak değilse veya ziyaret edilmişse 0 dön
    if (map[row][col] != resource || map[row][col] < 0)
        return 0;

    // Başlangıç noktasını ziyaret edilmiş olarak işaretle ve kuyruğa ekle
    map[row][col] = -1 * map[row][col];
    q.push({row, col});
    int size = 1; // Koloni boyutunu başlat

    // BFS döngüsü
    while (!q.empty()) {
        pii current = q.front(); // Kuyruktan bir düğüm al
        q.pop();

        // Dört ana yönü tanımla
        vector<pii> directions = {{0, 1}, {0, -1}, {1, 0}, {-1, 0}};
        for (auto dir : directions) {
            int newRow = current.first + dir.first;
            int newCol = current.second + dir.second;
            fix_row_col(map, newRow, newCol); // Harita sınırlarını tekrar düzelt

            // Eğer komşu hücre kaynak tipine uygunsa ve daha önce ziyaret edilmemişse
            if (map[newRow][newCol] == resource && map[newRow][newCol] > 0) {
                map[newRow][newCol] = -1 * map[newRow][newCol]; // Komşuyu ziyaret edilmiş olarak işaretle
                q.push({newRow, newCol}); // Kuyruğa ekle
                size++; // Koloni boyutunu arttır
            }
        }
    }

    return size; // Toplam koloni boyutunu dön

  /* END YOUR CODE HERE */
}
```

Pseudo Code of BFS Function

Function BFS(map: 2D vector of integers, row: integer, col: integer, resource: integer) -> integer

Create an empty queue Q

Call FixRowCol(map, row, col)

```
    If map[row][col] is not equal to resource OR map[row][col] is less than 0 Then
        Return 0
    End If

    Enqueue (row, col) into Q
    Mark map[row][col] as visited by negating its value (map[row][col] = -1 *
map[row][col])

    Set size to 1

    While Q is not empty Do
        (current_row, current_col) = Dequeue Q
        For each neighbor v of (current_row, current_col) Do
            Call FixRowCol(map, neighbor_row, neighbor_col)
            If map[neighbor_row][neighbor_col] is equal to resource AND
map[neighbor_row][neighbor_col] is greater than 0 Then
                Enqueue (neighbor_row, neighbor_col) into Q
                Mark map[neighbor_row][neighbor_col] as visited
                size = size + 1
            End If
        End For
    End While

    Return size
End Function
```

Time Complexity: O(N*M) in the worst case, where N is the number of rows and M is the number of columns in the map. This is because each cell is visited at most once.

Best Case: The starting cell does not contain the target resource or is already visited, leading to an immediate return. Time Complexity: Ω (1).

Average Case: Typically explores a subset of the grid to find all connected cells with the target resource, scaling with the number of cells visited. Time Complexity: Θ (N), where N is the number of cells visited, which could be less than the total number of cells.

In the BFS function, the algorithm starts by marking the starting cell as visited and enqueues it, initiating the size count of the colony with 1 to include the current cell. It then enters a loop where it dequeues a cell from the queue and explores its neighbors in all four cardinal directions: north, south, east, and west. For each neighboring cell that matches the resource type and hasn't been visited, the algorithm marks it as visited by negating its value, enqueues it, and increments the colony size counter. This process continues until no unvisited cells remain in the queue, ensuring a level-by-level exploration of the colony. The function ultimately returns the total size of the colony, which includes all connected cells sharing the same resource type, effectively calculating the colony's size by aggregating the individual cells discovered through this breadth-first search approach.

top_k_largest_colonies Funtion

```cpp
vector<pair<int, int>> top_k_largest_colonies(vector<vector<int>>& map, bool useDFS, unsigned int k)
{

    auto start = high_resolution_clock::now();      // Start measuring time

    /* START YOUR CODE HERE */
    vector<pair<int, int>>  colonySizes;

    if (!map.size())
        return (vector<pair<int, int> >());
    for (size_t i = 0, r = map.size(); i < r; i++)
    {
        for (size_t j = 0, c = map[0].size(); j < c; j++)
        {
            int resource = map[i][j];
            if (resource < 0)
                continue ;
            colonySizes.push_back({ useDFS ? dfs(map, i, j, resource) : bfs(map, i, j, resource), resource });
        }
    }
    /* END YOUR CODE HERE */

    auto stop = high_resolution_clock::now();                       // Stop measuring time
    auto duration = duration_cast<nanoseconds>(stop - start);       // Calculate the duration
    cout << "Time taken: " << duration.count() << " nanoseconds" << endl;

    /* START YOUR CODE HERE */
    sort(colonySizes.begin(), colonySizes.end(), customComparator);
    size_t  colony = min(static_cast<size_t>(k), colonySizes.size());
    vector<pair<int, int>>  top_k(colonySizes.begin(), colonySizes.begin() + colony);
    return (top_k);
    /* END YOUR CODE HERE */
}
```

Pseudo Code of top_k_largest_colonies Function

```
FUNCTION top_k_largest_colonies(map, useDFS, k)
    START timer

    INITIALIZE colonySizes as empty vector of pairs

    IF map is empty THEN
        RETURN empty vector of pairs

    FOR each row i in map DO
        FOR each column j in map DO
            SET resource to map[i][j]
            IF resource is non-negative THEN
                IF useDFS THEN
                    ADD pair(dfs(map, i, j, resource), resource) to colonySizes
                ELSE
                    ADD pair(bfs(map, i, j, resource), resource) to colonySizes
                END IF
            END IF
        END FOR
    END FOR

    STOP timer
    CALCULATE duration as the difference between stop and start timer
    PRINT duration in nanoseconds
```

```
    SORT colonySizes using customComparator

    SET colonyCount to minimum of k and size of colonySizes
    INITIALIZE top_k with first colonyCount elements of colonySizes

    RETURN top_k
END FUNCTION
```

Time Complexity: In this case, the total time complexity is derived from the combination of the DFS/BFS search and the sorting complexities: O(N*M + C log C).

O(N*M) represents the time required to visit all cells in the map, where N is the number of rows and M is the number of columns.

O(C log C) corresponds to the time needed to sort the sizes of the discovered colonies, with C being the total number of colonies identified. Because std::sort() function typically uses introsort algorithm. Speaking of which, worst case sorting may degreade to O(C^2).

Here, since N*M is typically larger than C (the number of discovered colonies), the overall complexity is primarily dependent on the size of the map and the efficiency of the DFS/BFS search operation. However, the O(C log C) term can become significant with a large number of colonies.

Best Case: The map is empty, leading to an immediate return. Time Complexity: Ω (1). This scenario bypasses the need for any search, sorting, or significant computation.

Average Case: The function's complexity mainly depends on the total area covered by all colonies and the sorting of colony sizes. Assuming the map is partially filled with colonies of varying sizes, and the searching algorithm (DFS or BFS) explores each cell at most once, the average complexity is Time Complexity: Θ (R*C + N log N), where R*C represents the total number of cells (R rows and C columns) explored by either DFS or BFS, and N log N represents the sorting of colony sizes, with N being the number of identified colonies.

In the top_k_largest_colonies function, the algorithm begins by timing its execution to evaluate performance. It then iterates over each cell of the provided map, ignoring cells with negative resources, as these are considered already visited so invalid. For each valid cell, it calculates the size of the colony (group of contiguous cells with the same resource type) it belongs to, using either a Depth-First Search (DFS) or Breadth-First Search (BFS) algorithm, based on the useDFS flag. Each colony's size and its resource type are paired and added to a list.

Once all cells have been processed, the execution time is recorded, and the list of colony sizes is sorted in descending order of size, with ties broken by the resource type's ascending order. The algorithm then selects the top k largest colonies from this sorted list. If the number of colonies found is less than k, it selects all available colonies. This method efficiently identifies and ranks the largest resource-rich areas within the map, returning a vector of pairs that represent the size and resource type of each top colony found. The entire process is monitored for execution time, emphasizing the algorithm's efficiency.

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

In my approach, I optimize for space complexity by managing discovered nodes directly within our in-memory data structure. Specifically, I updated visited nodes by multiplying by -1, directly modifying the map's content. This method eliminates the need for a separate list to track discovered nodes, thereby conserving memory. At the same time, it's important to note that this strategy allows user to keep data values as their negative values without completely losing the original data integrity, as it alters the data in place. While this decision effectively reduces the space required, it should have been aware of its implications on the data's original state and be ensured that this trade-off aligns with the project's objectives and requirements.

• How does the map size affect the performane of the algorithm for finding the largest colony in terms of time and space complexity?

Time Complexity: The primary factor is the algorithm's need to check each cell in the map at least once. Therefore, the time complexity directly correlates with the map's dimensions, resulting in a complexity of O(N*M), where N is the number of rows and M is the number of columns. This means that the time required to find the largest colony increases proportionally with the map's size.

Space Complexity: For DFS, the space complexity can go up to O(n*m) in the worst case due to recursion, especially if it is exploring a large connected component. For BFS, the space complexity is O(max(n, m)) because it stores the frontier (nodes to be visited next) in a queue. In a dense map, the queue might hold a large portion of the map's cells at once. For large maps, especially those featuring extensive colonies, this could mean significant memory requirements, but the primary factor remains the map's total size.

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

DFS explores as far as possible along each branch before backtracking, which can be efficient for deeply nested colonies but might lead to higher stack usage. Conversely, BFS explores neighbors level by level, which can quickly identify nearby large colonies but might increase queue usage due to broader exploration. In this project, DFS might be slightly faster for maps with large, compact colonies, while BFS could be advantageous for maps with dispersed, equally-sized colonies due to its uniform search pattern.

DFS can be more memory efficient in sparse maps where the call stack doesn't grow too large. It's also easier to implement recursively. However, in maps with large connected components, the stack size can grow significantly, affecting memory usage. In BFS memory usage can be a concern for dense maps, as the queue can grow large.

For the purpose of finding the largest colony, both DFS and BFS will explore all reachable cells of a colony. However, the difference comes in the order of exploration and memory usage. If the map has many large connected components, BFS might use more memory than

DFS due to queue size. In terms of execution time, both methods will have to explore all parts of the map, so the difference might not be significant unless the structure of the map significantly favors one method's exploration pattern over the other.