BLG 336E Analysis of Algorithms 2 Assignment 1

Due: Sunday, March 24, 2023

Lecturer: Assoc. Prof. Dr. Mehmet Baysan CRN: 21349

Fatih Baskın
150210710

Contents

1 Code Explaination

1.1 BFS

```
Algorithm BFS(map, row, col, resource)

  // map: 2D vector representing the map (0-based indexing)
  // map[i][j] = -1 means already visited

  row_max <- size(map)
  col_max <- size(map[0])

  // Handling edge cases
  IF size(map) = 0
    RETURN 0 // Empty map
  IF row < 0 or row >= row_max or col < 0 or col >= col_max
    RETURN 0 // Starting cell out of bounds
  IF map[row][col] != resource
    RETURN 0 // Current cell doesn't match resource type
  IF map[row][col] = -1
    RETURN 0  // Already visited, no colony here

  colony_size <- 0
  // Queue for BFS exploration
  Q <- empty queue
  // Mark the starting cell as visited (using -1)
  map[row][col] <- -1
  Enqueue(Q, (row, col))

  // Loop will visit all the cells if all cells are valid
  // Complexity: O(n x m) where n is number of rows and m is number of columns
  WHILE not isEmpty(Q)
    // Dequeue the current cell and increment the colony size
    // If a cell is in the queue, it is the only time we are encountering it in the queue
    // Since the cells are marked prematurely, no need to mark the current cell as visited
    (current_row, current_col) <- Dequeue(Q)
    colony_size <- colony_size + 1

    // Get valid neighbor coordinates (up, down, left, right)
    up <- GetUpCoordinate(current_row, current_col, row_max, col_max)
    down <- GetDownCoordinate(current_row, current_col, row_max, col_max)
    left <- GetLeftCoordinate(current_row, current_col, row_max, col_max)
    right <- GetRightCoordinate(current_row, current_col, row_max, col_max)

    // Enqueue valid and unvisited resource neighbors
    // Mark them as visited prematurely (using -1)
    IF map[up.row][up.col] != -1 AND map[up.row][up.col] = resource
      Enqueue(Q, up)
      map[up.row][up.col] <- -1
    IF map[down.row][down.col] != -1 AND map[down.row][down.col] = resource
      Enqueue(Q, down)
      map[down.row][down.col] <- -1
    IF map[left.row][left.col] != -1 AND map[left.row][left.col] = resource
      Enqueue(Q, left)
      map[left.row][left.col] <- -1
    IF map[right.row][right.col] != -1 AND map[right.row][right.col] = resource
      Enqueue(Q, right)
      map[right.row][right.col] <- -1

  RETURN colony_size
```

The code complexity is $O(n \times m)$ where n is the number of rows and m is the number of columns. **One important assumption is, that there is no resource as -1**, therefore -1 is used for marking a node as visited.

For performance reasons, my BFS implementation marks the enqueued nodes as visited (-1) prematurely, but before enqueuing them it checks for being visited or not and for being valid. Otherwise, a cell may be enqueued into the queue more than one time (on average two times) and it hampers the performance of the BFS. But on the other hand, it makes the code a little bit confusing.

1.2 DFS

```
ALGORITHM DFS(map, row, col, resource)
  // map: 2D vector representing the map (0-based indexing)
  // map[i][j] = -1 means already visited

  row_max <- length(map)
  col_max <- length(map[1])

  // Handle base cases (early returns):
  IF row_max = 0 OR col_max = 0
   RETURN 0 // Map is empty
  IF row < 0 OR row >= row_max OR col < 0 OR col >= col_max
   RETURN 0 // Starting cell outside map
  IF map[row][col] != resource
   RETURN 0 // Starting cell is not the resource type
  IF map[row][col] = -1
   RETURN 0  // Starting cell already visited, avoid revisit

  // Mark current cell as visited and initialize colony size as 1 (current cell)
  map[row][col] <- -1  // Change value to -1 to indicate visited
  colony_size <- 1

  // Explore neighboring cells recursively:
  up <- GetUpCoordinate(row, col, row_max, col_max)
  down <- GetDownCoordinate(row, col, row_max, col_max)
  left <- GetLeftCoordinate(row, col, row_max, col_max)
  right <- GetRightCoordinate(row, col, row_max, col_max)

  // Explore valid neighbors (up, down, left, right) with recursive calls
  // DFS will visit all the cells if they are the correct type
  // Complexity: O(n x m) where n is number of rows and m is number of columns
  IF map [up.row][up.col] != -1 AND map [up.row][up.col] = resource
    colony_size <- colony_size + DFS(map, up.row, up.col, resource)
  IF map[down.row][down.col] != -1 AND map[down.row][down.col] = resource
    colony_size <- colony_size + DFS(map, down.row, down.col, resource)
  IF map[left.row][left.col] != -1 AND map[left.row][left.col] = resource
    colony_size <- colony_size + DFS(map, left.row, left.col, resource)
  IF map[right.row][right] != -1 AND map[right.row][right.col] = resource
    colony_size <- colony_size + DFS(map, right.row, right.col, resource)

  RETURN colony_size
```

The code complexity is $O(n \times m)$ where n is the number of rows and m is the number of columns. **One important assumption is, that there is no resource as -1**, therefore -1 is used for marking a node as visited.

The code recursively walks around the grid if the neighbors are valid. One problem with this approach is, that it creates a very deep call stack, which might make the stack overflown in large grids. One way to solve this problem is to implement this code using a stack, iteratively, very similar to the BFS code while the only difference is the usage of a stack instead of a queue.

1.3 Top K Largest Colonies

```
ALGORITHM TopKColonies(map, useDFS, k)
  // map: 2D vector representing the map (0-based indexing)
  // useDFS: flag for whether using DFS or BFS

  // Max heap for storing colony size (larger first) and resource type (smaller first)
  CREATE max_heap colonies with comparison function CompareColonies

  // Find all colonies in the map
  FOR EACH row IN map DO
   FOR EACH col IN row DO
    IF map[row][col] != -1 // Skip visited cells
      colony_size <- 0
      resource_type <- map[row][col]
      // Call DFS or BFS depending on flag
      IF useDFS
        colony_size <- DFS(map, row, col, resource_type)
      ELSE
        colony_size <- BFS(map, row, col, resource_type)
      // If the visited cell is a valid cell with colony size at least 1
      IF colony_size > 0
        colonies.Enqueue((colony_size, resource_type))

  // Find top-k largest colonies
  top_k_colonies <- empty list
  WHILE colonies IS NOT empty AND top_k_colonies.size < k DO
    (colony_size, resource_type) <- colonies.Dequeue()
    top_k_colonies.Append((colony_size, resource_type))

  RETURN top_k_colonies
```

Each of the BFS and DFS functions has a time complexity of $O(n \times m)$. But the worst-case complexity of this function is $O((n \times m)log(n \times m))$ where n is the number of rows and m is the number of columns. Each cell in the 2D matrix (vector) may be a different type of resource and both DFS and BFS functions would have constant time since they would not be able to advance further of their starting cell. There would be $n \times m$ function calls to either DFS or BFS and since adding an element to the heap has $O(log(n))$ complexity (n stands for the number of elements in the heap) therefore it is possible to say this operation has $O((n \times m)log(n \times m))$ complexity. But, since we are giving elements one by one into the heap, initial steps have very low time complexity, therefore on average cases (the number of colonies is limited) this function has $O(n \times m)$ time complexity.

1.4 Helper Functions

```
// Define a comparison function to prioritize colonies in a max_heap
DEFINE comparison_function CompareColonies:
  // Compare two colonies for sorting in descending order
  FUNCTION Compare(colony_a, colony_b):
  IF colony_a.size == colony_b.size
    // If sizes are equal, prioritize smaller resource types
    RETURN colony_a.resource_type > colony_b.resource_type
  ELSE
    // Prioritize larger colony sizes
    RETURN colony_a.size < colony_b.size
```

```
// (wrap around boundaries for up, down, left, right)
FUNCTION GetUpCoordinate(coordinate, row_max, col_max)
  row = (coordinate.row + row_max - 1) % row_max  // Wrap around for up
  col = coordinate.col
  RETURN (row, col)

```

```
FUNCTION GetDownCoordinate(coordinate, row_max, col_max)
  row = (coordinate.row + 1) % row_max // Wrap around for down
  col = coordinate.col
  RETURN (row, col)

FUNCTION GetLeftCoordinate(coordinate, row_max, col_max)
  row = coordinate.row
  col = (coordinate.col + col_max - 1) % col_max // Wrap around for left
  RETURN (row, col)

FUNCTION GetRightCoordinate(coordinate, row_max, col_max)
  row = coordinate.row
  col = (coordinate.col + 1) % col_max // Wrap around for right
  RETURN (row, col)
```

Custom comparator struct defines comparison characteristics of the heaps (priority queues). The implementation of them is language-specific. In `C++`, it is possible to overload the `()` operator of a struct and use that struct as a comparator.

These helper functions run in constant time, $O(1)$. They use modular arithmetic to move circularly. In the cases where I do subtraction, I add `col_max` or `row_max` since modulo of negative values are problematic, to avoid such problems I add these values, ensuring I won't get a negative value. These values are effectively 0 when taking modulo.

2 Map Size and Performance

2.1 Complexity of BFS and DFS concerning Map Size

Map size directly indicates the number of cells available to visit in the grid. The performance of both DFS and BFS algorithms increases linearly with the number of available nodes (cells in the grid) available. Therefore, the performance of searching the grid is linearly proportional to the map size. If we call the number of rows n and the number of columns m, then the total number of cells available on the grid (planets in the map) is $n \times m$, and since DFS and BFS algorithms may visit each grid cell then their time complexity is $O(n \times m)$.

The space complexity of the DFS code is also $O(n \times m)$ since it is possible to check every single cell in a very deep call stack if the whole map consists of the same type of planets. On the other hand, in BFS, since it moves layer by layer, it grows in a diamond-cross-shaped pattern, therefore the maximum space held for the neighbors in the queue is proportional to the largest possible diagonal in the map. Therefore it is proportional to $\sqrt{n^2 + m^2}$ (diagonal length in the 2D map) but since $\sqrt{n^2 + m^2} < \sqrt{n^2 + m^2 + 2 \times n \times m} = n + m \; for \; n, m > 0$, it is possible to claim the space complexity, which is the number of items held in the queue, has the complexity of $O(n + m)$.

2.2 Complexity of Finding K^{th} Largest Nodes

There is another thing to consider here, in the code explained in the **section 1.3**, the implementation of finding the k-largest colonies uses a priority queue, a max-heap. Since insertions take $O(log(n))$ time, for n being the number of elements in the heap, if all the planets on the map have different types of resources, then there would be $n \times m$ insertions where n stands for the number of rows and m stands for the number of columns. In this case, both BFS and DFS algorithms will run in constant time because they can't traverse the neighbors. Overall, in this worst case, the code would have $O((n \times m))log(n \times m)$ time complexity and space complexity of $O(n \times m)$, all the planets (single size colonies) and their respective resources being in the heap.

On average, adding elements to the priority queue does not have this large overhead since there are a limited number of colonies. Therefore, the overhead of BFS & DFS would be the defining factor on the runtime.

3 Maintaining Visited Nodes

The visited nodes must be maintained otherwise, both BFS & DFS algorithms would run indefinitely since they will not know which nodes are visited and which are not. **Since the homework had limitations for the implementations**

of the functions, to keep track of which nodes are visited, I have marked visited nodes as -1 on the original map, 2D grid. By keeping track of visited (or discovered) nodes, DFS and BFS would terminate and they would not visit the same node twice.

4 Performance Comparison

BFS and DFS algorithms have the same asymptotic time complexity, $O(n \times m)$ where n stands for number of rows and m is the number of columns in the grid. Since they have the same asymptotic time complexity, it is expected that their runtimes are similar. Only difference they have is their space complexities, which is explained in **section 2.1**. Space complexity of BFS is $O(n + m)$ and DFS is $O(n \times m)$.

It is said that BFS and DFS have the same asymptotic time complexity, however I was expecting that DFS might work slower due to DFS being recursive, with the intuition that overhead of doing function calls make it slower than the iterative BFS. But, the real world data says the otherwise, in fact for this homework, on average, DFS have worked faster. I have tested both DFS and BFS in the five maps provided and gathered their results. I have made them run for top 1000 colonies, which is larger than the number of cells in the grid (planets on the map). The results are given in the **table 1** as follows:

Map	DFS Runtime (ns)	BFS Runtime (ns)
Map 1	26931	51066
Map 2	5711	11452
Map 3	4178	7804
Map 4	105760	211901
Map 5	1002	972

Table 1: Runtimes of DFS and BFS

On average, it seems like BFS has taken 2 times longer than the DFS. I suspect inefficiency of using the queue data structure is causing this, since DFS is using the program's stack memory. Using the stack memory of the program might be faster due to program stack being a very large space and advancing the stack pointer and copying function parameters being a relatively short task.On the other hand, queue might be dealing with dynamic memory allocation, which requires some system calls and those are not a simple task.

According to my research, C++ queues use a data structure called deque [1], and deque does dynamic memory allocation for its elements[2].

References

[1] cppreference.com. "Std::queue." (2024), [Online]. Available: `https://en.cppreference.com/w/cpp/container/queue` (visited on 03/20/2024).

[2] cppreference.com. "Std::queue." (2023), [Online]. Available: `https://en.cppreference.com/w/cpp/container/queue` (visited on 03/20/2024).