BLG 336E- Analysis of Algorithms II Homework 1 Report

Özkan Gezmiş 150200033

1)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.

a)pseudo-code:

```
#define VISITED_ELEMENT_RESOURCE -1000 // macro value for make elements of map visited
class Node // class to represent the elements of the map
   public:
   int resource;
   pair<int,int> index;

top_k_largest_colonies(map, useDFS, k):
create k sized vector of pair<int,int> name largest_colonies
for each row in the map:
    for each column in the row:
        if resource_type of element == VISITED_ELEMENT_RESOURCE:
            continue  to next element
        else:
            if useDFS == true:
                size_of_component = dfs(map, row, col, resource_type)
            else
                size_of_component = bfs(map, row, col, resource_type);
            if size_of_component > smallest element of largest colonies:
                delete smallest element from vector and add new component size
        sort(largest_colonies)
return largest_colonies


bfs(map, row, col, wanted_resource):
if given element's resource != wanted_resource
    return 0
create queue of Nodes named bfsQueue
push first element to queue and make its resource type VISITED_ELEMENT_RESOURCE
while stack is not empty:
    current element = bfsQueue.front()
    bfsQueue.pop()
    component_size++
    row = currentElement.index.first
    col = currentElement.index.second
    push_neigbors_to_queue(row, col, length_of_map, height_of_map, wanted_resource, map,bfsQueue)
return component_size


dfs(map, row, col, wanted_resource):
if given element's resource != wanted_resource
    return 0
create stack of Nodes named dfsStack
push first element to stack and make its resource type VISITED_ELEMENT_RESOURCE
while stack is not empty:
    current element = dfsStack.front()
    dfsStack.pop()
    component_size++
    row = currentElement.index.first
    col = currentElement.index.second
    push_neigbors_to_stack(row, col, length_of_map, height_of_map, wanted_resource, map, dfsStack)
return component_size
```

```
push_neigbors_to_queue(row, col, lenght, height, resource, map, bfsQueue):
        if left left, right, bottom, top elements of current element are not added to the queue:
                add left, right, bottom, top elements to the queue independently
                make newly added elements' resource type VISITED_ELEMENT_RESOURCE
```

```
push_neigbors_to_stack(row, col, lenght, height, resource, map, dfsStack):
        if left left, right, bottom, top elements of current element are not added to the stack:
                add left, right, bottom, top elements to the stack independently
                make newly added elements' resource type VISITED_ELEMENT_RESOURCE
```

b) Complexity of the functions:

push_neigbors_to_stack() and *push_neigbors_to_queue()* functions check at most 8 elements and push at most 4 elements to the queue or the stack. Therefore, they run on **O(1)**.

dfs() and *bfs()* functions have same complexity. In general this is O(edge_count + vertex_count). In our case each vertex has 4 edges so it turns out to O(5* vertex_count).

But this is not the whole element number, this is just the size of the connected_component. We can say the complexities are **O(vertex_count)** which is also equal to **O(row*column)**, but in amortized time complexity will be much more smaller.

top_k_largest_colonies() function iterates over the whole map. In each iteration, if the element is not visited in other turns, calls the bfs() or dfs() functions. And sorts elements of the vector. For sorting the general complexity is O(n*logn) but in our case we have just limited number of elements in the vector so we can say complexity for sorting is O(k*logk) which is actually O(1). In theory if I use priority queue adding element to queue would be just O(logn) but in that case keeping track of the element number would be difficult, since we are asked to find k pieces largest component.
Since, for dfs() and bfs() we have complexity of O(vertex_count) and we iterate loop for row*column times which is also O(vertex_count), the total complexity is O(vertex_count^2). And if we say vertex_count is n the complexity would be **O(n^2)**. But if we consider k is a variable then we have extra O(k*logk); therefore, we can say that complexity of the algorithm is **O(k*logk*n^2).**

2) **Why should you maintain a list of discovered nodes? How does this affect the outcome of the algorithms?**

We need to have additional data structure to keep discovered nodes. Since If we don't add expanded neighbours to a list, we would lose the connections. Furthermore, it may cause to infinite loops, or if we don't use additional list, we may need to visit nodes more than once. For bfs and dfs we need to keep track of the component size, and if we visit nodes more than once this component size value probably wouldn't be true.

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

I found the complexity of *top_k_largest_colonies()* as **O(map_size^2)**. That means if we make map k times bigger our algorithm will be k^2 times slower.

For space complexity, we would require approximately k times bigger memory. Since, we use map as reference and just one map object is enough for the algorithm. dfs() and bfs() may use larger lists but compared to map size it would be insignificant. At the end space complexity of the algorithm is **O(map_size)**

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

Due to both dfs() and bfs() finding whole connected components at one iteration in terms of complexity, they have the same complexity. Both of them have O(vertex number + edge number) complexity. As we can see in the code, the time difference between these two algorithms is almost insignificant.