ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 336E

ANALYSIS OF ALGORITHMS
ASSIGNMENT 1

070170450 : MUHAMMED ENES DENİZ

SPRING 2022

Contents

1 Question 1

A) In the project, it is expected to bring a graph approach to the admiral sunk game. In this way, the game could be played with BFS and DFS algorithms.

To achieve the presented goals and expectations that is written in the project document I have come out with the following solution: By accepting each cell of the matrix, which is given as the game map, as a node, by understanding the neighborhoods of these nodes as edges, neighborhoods are found for each node and the algorithm works according to the priorities defined over these neighborhoods.

In order to achieve the purpose described above, I have defined 2 classes, one of them is the Node class and the other one is the GameBoard class. The Node class is designed to represent a cell in a 2D Array(matrix). Besides the attributes representing graph states as attributes, there is a variable defined to express the presence of ships in the node. At the same time, each node has a list within itself to keep its neighbors. The gameboard class represents the map that the player has. It contains a 2D vector of nodes and contains basic information for the player, such as the address of the player's first attack and the number and length of ships he owns. In addition, BFS and DFS algorithms are also available as member functions of this class. Finally, it has a queue holding nodes for the BFS method and a stack holding nodes for the DFS method.

```
//Create Vector<Vector<Node> >

for i in range row:
	for j in range column:
		create new Node;
		place ship if exists to the Node;
		rowVector.pushBack(Node);
	map.pushBack(rowVector);

// Place neighbors of the nodes into the vector that is kept in the node itself
	//According to the neighborhood situations, the push order is indicated as in pseudocode
for i in range row:
	for j in range column:
		if //UP LEFT CORNER
			node.neighborlist.push(Bot & Right Neighbor)
		else if //UP RIGHT CORNER
			node.neighborlist.push(Left & Bot Neighbor)
		else if //FIRST ROW EXCLUDED UP LEFT - RIGHT CORNERS
			node.neighborlist.push(Left & Bot & Right Neighbor)
		else if //DOWN LEFT CORNER
			node.neighborlist.push(Top & Right Neighbor)
		else if //DOWN RIGHT CORNER
			node.neighborlist.push(Top & Left Neighbor)
		else if //LAST ROW EXCLUDED DOWN LEFT - RIGHT CORNERS
			node.neighborlist.push(Top & Left & Right Neighbor)
		else if //FIRST COLUMN EXCLUDED LEFT UP - DOWN CORNERS
			node.neighborlist.push(Top & Bot & Right Neighbor)
		else if //LAST COLUMN EXCLUDED RIGHT UP DOWN CORNERS
			node.neighborlist.push(Top & Left & Bot Neighbor)
		else   //MIDDLE NODES
			node.neighborlist.push(Top & Left & Bot & Right Neighbor)
```

Figure 1: Game Board Preparation

If we examine the general flow of the program as some crucial parts depicted in figure 1: The input files given as the command line argument are read and a map (2D Vector) of the specified size is created for two players by defining GameBoard object for each player.The map is filled with nodes, taking into account the index values, the number of nodes kept in memory equal to the size of the map since they are created and stored in the 2D Vector. Then, the process of connecting the neighborhoods of the nodes is started. While connecting the neighbors of nodes, 8 conditions are checked and accordingly the neighbors are tied. Neighbors of nodes are kept into the vector that is present in the node itself. After these operations are performed for both players, the maps of the players are become ready. In the game phase, the players perform their first attacks on their opponent board according to the location they indicated in the input file. After that according to the algorithm type they specify in the input file, the game is played by the algorithm themselves in a turn based approach until one of the player's all ships are sunk.

```
//Game Play
       //Depending of the algo type BFS or DFS call happens
while player's have ship:
       call player1.DFS(p2) or player2.BFS(2)
       if Player2's all ships sunk
              player1 wins;
              break;

       call player2.DFS(p1) or player2.BFS(2)
       if Player1's all ships sunk
              player1 wins;
              break;
```

Figure 2: Turn Based Game Play

The game is designed to be played as turn based it could be observable from the above graph.

```
GameBoard::DFS(GameBoard& competitor):

        while(top of stack is a visited node):
                pop stack;

        top of stack is placed in *temp_node;
        pop stack
        set *temp_node as visited
        If *temp_node has ship
                Set node's ship available to false
                Decrease opponent number of ship by one
                If opponent has no ship left
                        Return flag to indicate game finished

        j = length(neighborList) - 1;
        for each element i in the *temp_node neighbors:
                If temp_node->neighborlist[j] is not visited:
                        Push neighborlist[j] to stack;
                j--

         Return flag to indicate game is not finished


GameBoard::BFS(GameBoard& competitor):
        top of queue is placed in temp *node;
        pop queue
        If Temp *node has ship
                Set node's ship available to false
                Decrease opponent number of ship by one
                If opponent has no ship left
                        Return flag to indicate game finished

        for each element i in the temp *node neighbors:
                If node->neighborlist[i] is not visited and is not already pushed to queue:
                        Push neighborlist[i] to queue;
                        Set neighborlist[i] as pushed already to queue

        Return flag to indicate game is not finished
```

Figure 3: BFS and DFS Algorithms

As could be observable from the pseudo code's of the BFS and DFS the time complexity of the algorithms depends mainly on the visiting a node. To examine the BFS and DFS time complexity let's assume an input graph G(V,E). The operations handled when we visit a node takes O(1) time as it could be observable from the pseudocode. Since we could only visit a node only once, the time that is require to process each node is equal to the node number V. When a node is visited it's neighbors list(adjancency list) is examined and this scanning procedure will at most take E times. Thus, the time complexity of the algorithms are O(V+E) which is linear time.

For the space complexity the same manner could be applied. In my presented solution, I have treated each cell of the matrix as a node and created the full graph in the first place, then when I visit nodes, I calculated the potential attack location by neighbors(edges) and stored them in stack or queue. Thus, the space complexity of the algorithms O(V+E). To be more precise about the memory complexity when constructing the graph the node is required is equal to the square of the ,size, row or column.

2 Question 2

Game 2 DFS	*The Number of Visited Node*	*The Number of Nodes Kept in Memory*	*Running Time*
Player1	42	49	6
Player2	41	49	6

Game 3 BFS	*The Number of Visited Node*	*The Number of Nodes Kept in Memory*	*Running Time*
Player1	48	49	7
Player2	47	49	7

Figure 4: Game 2 and Game 3 Results

The numbers of nodes kept the in the memory is same for the both case since we did not change the board size. It is observable from the figure 4 that DFS game is finished earlier than the BFS game. The linear time complexity calculation seems to be hold O(V+E) or simply O(n).

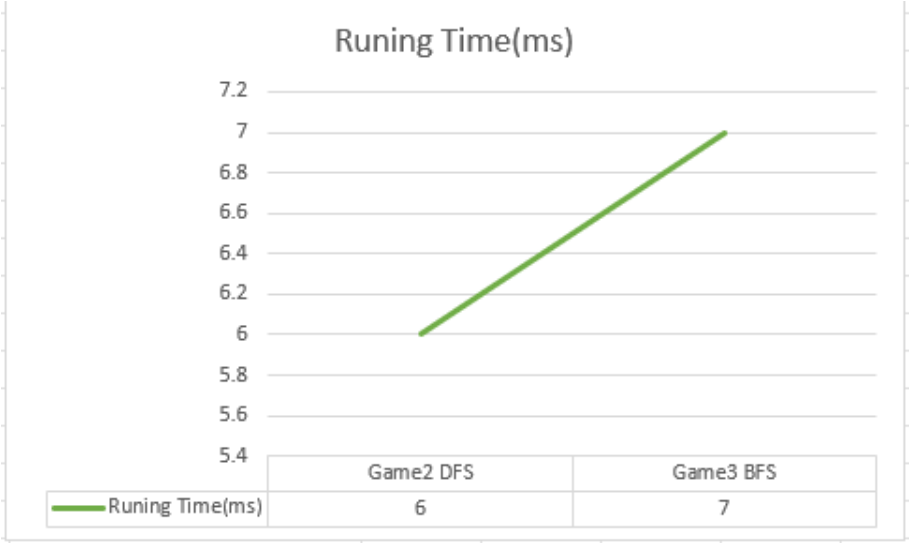

Figure 5: Game 2 and Game 3 Results

As calculated the time complexity ,O(V+E), of the DFS and BFS bounded by linear time O(n). When the number of the nodes increases linearly, running time increases with the same manner.

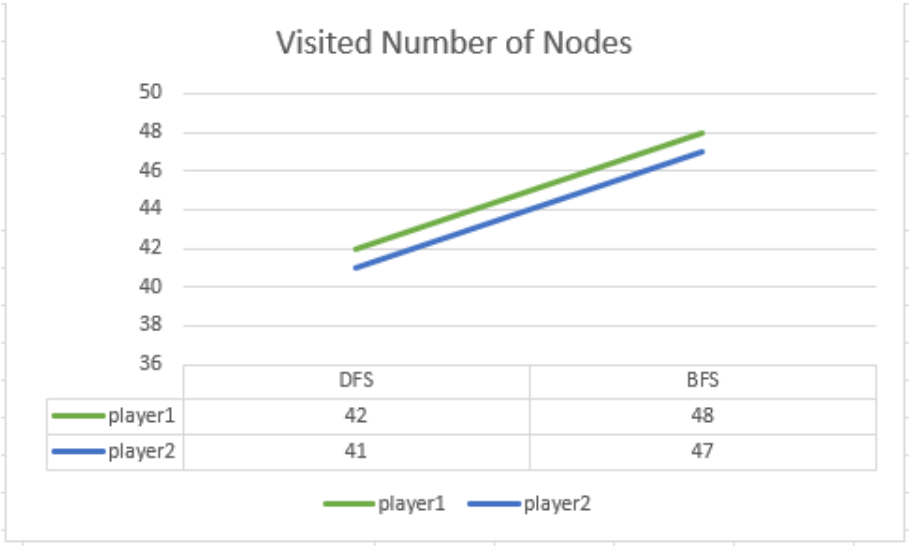

Figure 6: Game 2 and Game 3 Results

The performance of the algorithms depends on the given ship coordinates in the matrix and also starting location of attack. When the enemy ships coordinates are given away from the source DFS had more advantage to BFS since DFS is could easily access depths where BFS considers the all neighbors first. To sum up, starting location and given input matrix affects the performance of the algorithms. For the presented case DFS is faster than the BFS

3 Question 3

When I visit a node, I mark that node as visited, and in this way, I prevent going to that node again. Since, I treat the cells of map as nodes, if I do not mark and specify the nodes I have visited before and visit them again my implementation cause results deviated from the expected. Moreover If I do not specify the visited nodes complexity of my algorithm will increase since I will make redundant operations. In short, I marked visited nodes to make sure my algorithm work flawlessly and to not increase my complexity with redundant operations.

4 Question 4

At the point where the size of the map is increased but the number of ships is kept the same, our memory complexity increases because we have a situation where we increase the board size in other words number of nodes possible to be visited. According to my approach the nodes are considered as matrix cells so when the board size increases also my node number increases , node number = square of row or column, and this affects the

memory complexity. Apart from this, we can say that the size of the queue and stack where the algorithms store their possible moves will also increase. To summarize, if the map is enlarged, there will be more possible moves and more nodes will be visited more nodes will be pushed into the stack and queue and initially more nodes will be created because of the board size thus complexity will increase.