**Department of Computer Engineering**

# BLG 336E
# Analysis of Algorithms II
# Project 1 Report

**Spring 2019**

**Muhammed Raşit EROL**

**150150023**

## 1 INTRODUCTION

In this project, it is required to implement BFS and DFS algorithms in order to solve the given gold miners problem. The requirement of the solution for given matrix is explained in the project pdf with details. In order to solve this graph traversal problem, it is implemented BFS and DFS search functions are used and details of that implementation is explained following parts.

## 2 IMPLEMENTATION

In this project, it is used two class which are Graph and Node in order to traverse on the states of the graph. Graph is consisting of Nodes. The class structure can be seen in the code.

All operations like bfs and dfs functions are called from Graph class. Furthermore, root of the graph is kept with the Node pointer variable. In order to perform bfs and dfs search, for bfs, queue data structure and for dfs, stack data structure are used. Also, for output, number of visited nodes and number of nodes which is kept in the memory are stored in the Graph class. Moreover, for keeping visited Nodes, map data structure is used.

The Node class store the state of the matrix and its validation information. In order keep matrix in the map, matrix_string variable is used. Also, for creating proper Graph, all nodes keep the its children's addresses in the childs_ptr variable.

Each state has its input matrix and miners are placed in this matrix. After all children are created the validation control is made with the chech_node function. The node creation is made with the constructer of the Node class and if the matrix is valid then it is pushed the stack or queue by its algorithm type. For example, for root, all children nodes are created and valid ones are pushed the proper data structure and workflow of the program goes like that. Also, in the check_node function whether final state of the matrix is checked. If it is reached proper solution, then program is terminated with the finish_flag variable. DFS and BFS graph traversal is implemented like explained above.

The code compiled at ITU SSH server successfully like that:

```
g++ *.cpp –o algo -std=c++11

./algo bfs input_0.txt output_0.txt
```

## 3 ANALYSIS

The analyse of the code is explained following part. The complexity of the code depends on the used data structure in this project. All details of the functions with their complexities are explained below. The pseudo code of the bfs and dfs functions can be seen below.

Firstly, bfs function is analysed below.

```
bfs():
        Set the temp node Node *temp = NULL;
        While bfs_queque is not empty
                Set the temp to front element of the queque temp = bfs_queque.front();
                Function call bfs_create_childs(temp);
                If finish_flag is true
                        return;
                Endif
                Pop the queque bfs_queque.pop();

        EndWhile
```

```
bfs_create_childs(Node* node_ptr):
        For each element i on the row of matrix
                For each element j on the column of matrix
                        If the matrix[i][j] is “.”
                                Create node
                                Add the node to children vector
                                Increase counter of number of nodes in the memory
                                If Node is not in the visited map
                                        Add the node to the map
                                        Increase counter of number of visited
                                        If final state is reached (check function O(1))
                                                Set finish flag
                                                Return
                                        Endif
                                        If matrix is valid
                                                Push the node to queue
                                        Endif
                                Endif

                        Endif

                Endfor

        Endfor
```

The time complexity of a BFS algorithm depends directly on how much time it takes to visit a node. Since the time it takes to read a node’s value and pushing its children doesn’t change based on the node, we can say that visiting a node takes constant time, or, O(1) time. Since we only visit each node in a BFS tree traversal exactly once, the time it will take us to read every node really just depends on how many nodes there are in the tree. Thus, the time complexity of a breadth-first search algorithm takes linear time, or O(V+E), where V is the number of vertices in the tree and E is the number of edges of the tree.

Space complexity is similar to this. In the worst case situation, we could potentially be pushing all the nodes in a tree if they have all valid matrix, which means that we could possibly be using as much memory as there are nodes in the tree. If the size of the queue can grow to be the number of nodes in the tree, the space complexity for a BFS algorithm is also linear time, or O(V+E).

Shorty, all operation on the pseudo code is bounded by O(1); hence, the time complexity and space complexity depends on the number of summation of edges and vertices in the graph.

The dfs function is analysed below.

```
dfs():
      Set the temp node Node *temp = NULL;
      While dfs_stack is not empty
            Set the temp to front element of the stack temp = dfs_stack.top();
            Function call dfs_create_childs(temp);
            If finish_flag is true
                  return;
            Endif
            Pop the stack dfs_stack.pop();

      EndWhile
```

```
dfs_create_childs(Node* node_ptr):
      For each element i on the row of matrix
            For each element j on the column of matrix
                  If the matrix[i][j] is “.”
                        Create node
                        Add the node to children vector
                        Increase counter of number of nodes in the memory
                        If Node is not in the visited map
                              Add the node to the map
                              Increase counter of number of visited
                              If final state is reached (check function)
                                    Set finish flag
                                    Return
                              Endif
                              If matrix is valid
                                    Push the node to stack
                              Endif
                        Endif

                  Endif

            Endfor

      Endfor
```

The similar data structure is used in the dfs function. Instead of using queue, in the dfs algorithm stack data structure is used. Since the time complexity and space complexity is similar to the queue, using stack data structure is not change the both time and space complexity of the algorithm. Hence, similar to the bfs algorithm, dfs algorithm’s time and space complexity is O(V+E).

Furthermore, in the code, visited nodes are stored because there is possibly to visit visited nodes in the graph, which increases the time and space complexity of the code. Actually, if we do not keep track of the visited nodes then we increase the total visited node number, which is n and it affects the time and space complexity of our functions.

The comparison of the algorithms can be analysed below.

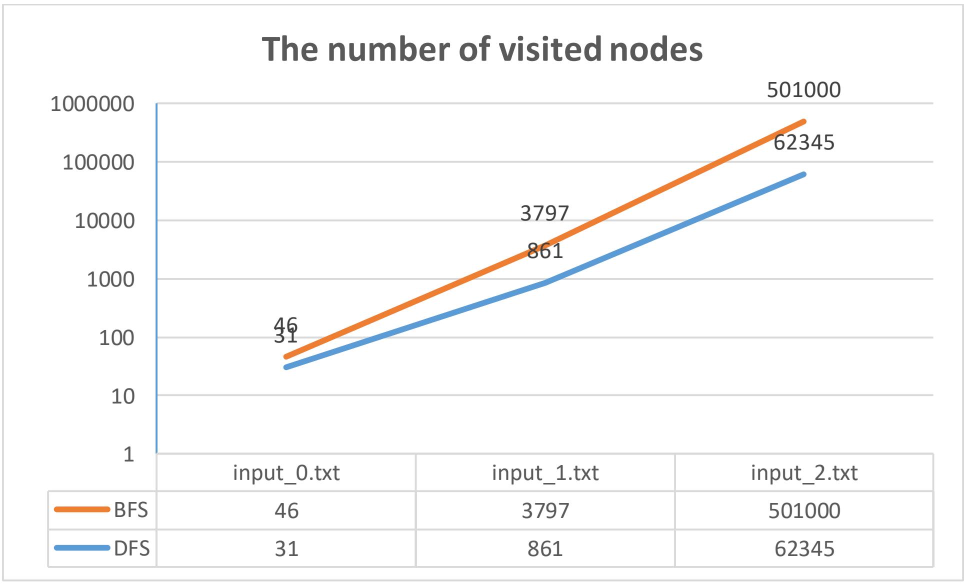
| BFS | input_0.txt | input_1.txt | input_2.txt | input_3.txt |
|---|---|---|---|---|
| The number of visited nodes | 46 | 3797 | 501000 | - |
| The maximum number of nodes kept in the memory | 56 | 5477 | 592534 | - |
| The running time. | ~0 | 0,03 | 6,21 | - |

*Table 1: BFS table*

| DFS | input_0.txt | input_1.txt | input_2.txt | input_3.txt |
|---|---|---|---|---|
| The number of visited nodes | 31 | 861 | 62345 | - |
| The maximum number of nodes kept in the memory | 32 | 1928 | 135712 | - |
| The running time. | ~0 | 0,01 | 1,48 | - |

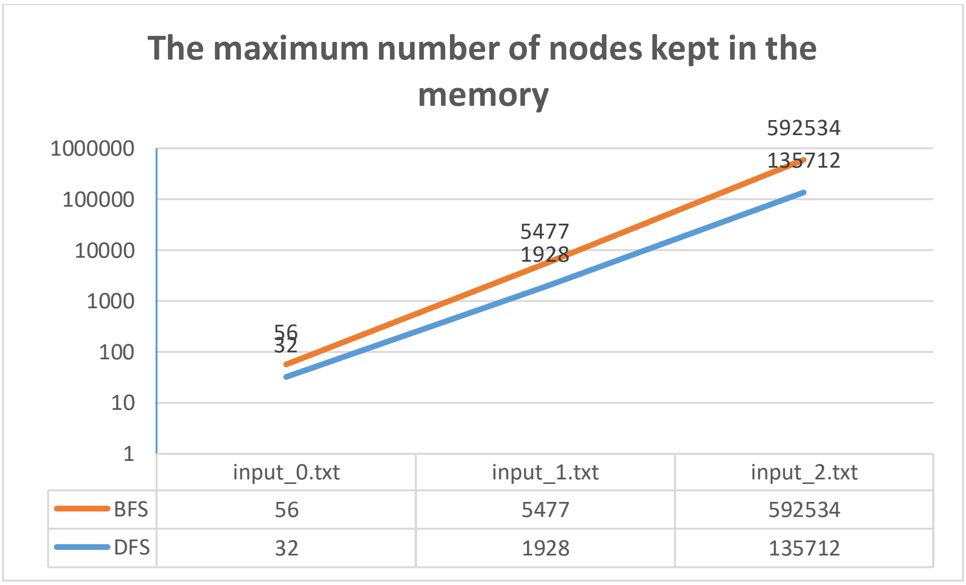
*Table 2: DFS table*

The graph of the Table 1 and Table 2 can be seen below.



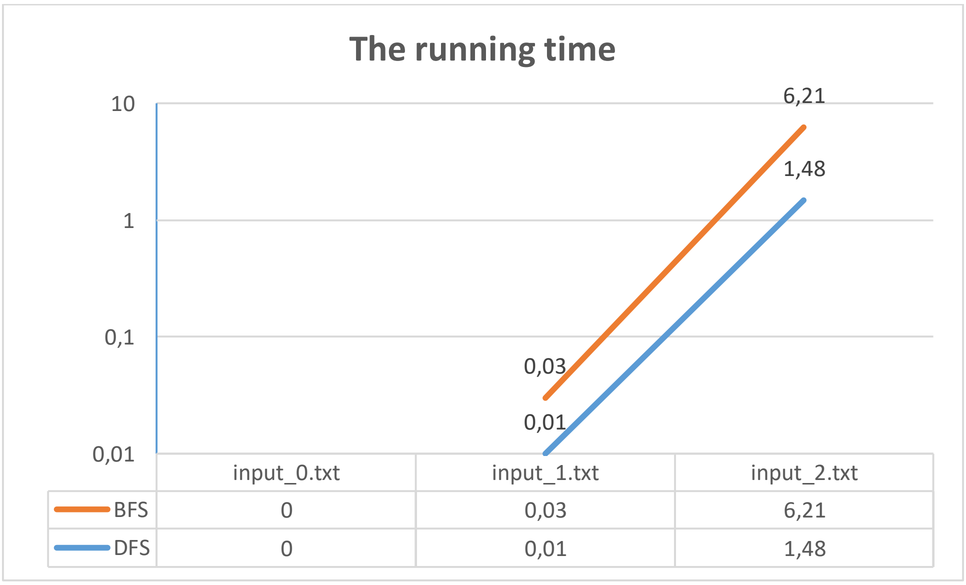
|  | input_0.txt | input_1.txt | input_2.txt |
|---|---|---|---|
| BFS | 46 | 3797 | 501000 |
| DFS | 31 | 861 | 62345 |

*Figure 1: The number of visited nodes*



| | input_0.txt | input_1.txt | input_2.txt |
|---|---|---|---|
| BFS | 56 | 5477 | 592534 |
| DFS | 32 | 1928 | 135712 |

*Figure 2: The maximum number of nodes kept in the memory*



| | input_0.txt | input_1.txt | input_2.txt |
|---|---|---|---|
| BFS | 0 | 0,03 | 6,21 |
| DFS | 0 | 0,01 | 1,48 |

*Figure 3: The running time*

In the Figure 1, Figure 2 and Figure 3, it can be seen that time complexity and space complexity of the bfs and dfs functions are linear time which is O(n). The performance of these algorithms depends on the given matrix. Because solution may appear in different level of the graph; hence, some matrixes bfs may work better or vice versa.

Furthermore, input_3.txt file is not tested, because, graph is huge for given matrix. Also, after some time during execution, given space for stack and queue become full and program is terminated with exception.