BLG336E Analysis of Algorithms 2

Project 1

Fatih Budak

150130006

In our code, firstly we have performed reading operation from input file and create a first node according to data in input file. Then, we create a graph and first node is connected to our graph. After that, According to parameter taken form console, our program is executed for BFS algorithm or DFS algorithm. Also we have implemented these functions recursively. Also, we write to output file each of node in the desired format. In addition to these information, as we have handled the cycle problem, we use a function called **comparefunction**. Thanks to this function, we do not produce a node previously produced. Therefore, we have checked the cycles.

BFS (Graph , Node , Output File)

for i ← 1 to the_number_of_blocks[Node]

if Direction[Node]=='h'

if i^{th} block can move one unit left

New_Node ← Node

İf Compare_function(Node, New_Node)==0

Queue ← New_Node

İf Final State is reached

Print Output File Function(New_Node)

return

if i^{th} block can move one unit right

New_Node ← Node

İf Compare_function(Node, New_Node)==0

Queue ← New_Node

İf Final State is reached

Print Output File Function(New_Node)

return

else

if i^{th} block can move one unit up

New_Node ← Node

İf Compare_function(Node, New_Node)==0

Queue ← New_Node

İf Final State is reached

Print Output File Function(New_Node)

return

if i^{th} block can move one unit down

```
                    New_Node ← Node
                    İf Compare_function(Node, New_Node)==0
                            Queue ← New_Node
                    İf Final State is reached
                            Print Output File Function( New_Node)
                            return

if empty[Queue]
        return
else if  size[Queue]
        Print Output File Function( Front [ Queue ] )
        Queue.pop()
        BFS( Graph , Node , Output File)
```

DFS (Graph , Node , Output File)

```
for i ← 1  to the_number_of_blocks[Node]
        if  Direction[Node]=='h'
                if   i^th  block can move one unit left
                        New_Node ← Node
                        İf Compare_function(Node, New_Node)==0
                                Stack ← New_Node
                        İf Final State is reached
                                Print Output File Function( New_Node)
                                return
                if   i^th  block can move one unit right
                        New_Node ← Node
                        İf Compare_function(Node, New_Node)==0
                                Stack ← New_Node
                        İf Final State is reached
                                Print Output File Function( New_Node)
```

```
                    return
        else
            if  i^th block can move one unit up
                New_Node ← Node
                İf Compare_function(Node, New_Node)==0
                    Stack ← New_Node
                İf Final State is reached
                    Print Output File Function( New_Node)
                    return
            if  i^th block can move one unit down
                New_Node ← Node
                İf Compare_function(Node, New_Node)==0
                    Stack ← New_Node
                İf Final State is reached
                    Print Output File Function( New_Node)
                    return

if empty[Stack]
        return
else if  size[Stack]
        Print Output File Function( Top [ Stack ] )
        Stack.pop()
        DFS( Graph , Node , Output File)
```

And finally, we print to console **the number of nodes generated** and **the maximum number of nodes kept in the memory** and **running time** for BFS and DFS.

We have used two class named **Node** and **Graph.**

Node Class:

```
class Node{

    private:
        int **matrix;
        char *directions;
        int **indises;
        int *sizes;
        int block_number;

    public:
        Node();
        void set_first_state();
        void build_matrix(int, int, int, char, int);
        void set_matrix(int, int, int, char, int);
        void set_indices(int, int, int);
        void set_directions(int, char);
        void set_sizes(int, int);
        int ** get_matrix();
        int ** get_indises();
        char * get_directions();
        int * get_sizes();
        int get_block_number();
        void set_block_number(int );
};
```

In Node class, we store five private member variables:

- **Matrix** variable is for box that includes blocks.
- **Direction** variable is to store only direction of blocks, ‘h’ for horizontal and ‘v’ for vertical.
- **Indises** variable is to store indises of blocks.
- **Sizes** variable is for size of blocks, like 2 unit or 3 unit.
- **Block_Number** is to store the number of blocks in the box.

We write also getter and setter functions private member variables. In addition to these function, we write two method:

- **set_first_state** function is used to allocate memory for indises, sizes and direction variables according to the number of block number.
- **Build_matrix** function is used to create first state node.

Graph Class:

```
class Graph{

    private:
        Node *root;

    public:
        queue <Node*> C1;
        stack <Node*> C2;
        vector <Node*> compare;
        vector<Node*>::iterator it;

        Node *children;
        Graph();
        void set_root(Node *);
        Node * get_root();
        void build_child(Node *);

};
```

In Graph class, we store one private four private member variables:

- **Root** is necessary to connect the first state to our graph.
- **Queue** is for BFS function
- **Stack** is DFS function
- İt is just a vector type iterator, is used in compare function.

We have written also getter and setter function for root private member variable. Moreover, thanks to **build_child** function, we create new nodes/states as we can reach the final state.

As we mentioned before, we prevent to occur the cycle in our graph with compare function. Briefly, each node that be produced is put in the vector and new nodes are compared the nodes in the vector. If a node in the vector is the same new node, we do not push queue for BFS or stack DFS. If not, we push queue for BFS or stack DFS. Therefore we have used extra memory for all generated nodes in vector. This situation increases the memory consumption of our program.