BLG336E ANALYSIS OF ALGORITHMS 2

Project 1 Report

Ece Naz Sefercioğlu
150130140

1)

How does your algorithm work? Write your pseudo-code.

BFS:

İnitiate State vector royals to store unique states

İnitiate 2D State vector L to contain layers

Create discovered array false in initialization

BFS edge tree empty

İnitiate idCounter to 1

Set the layer counter iy=0

Set win condition won to false

While L[iy] is not empty and won is false

    İnitiate L[iy+1]

    For each State in L[iy]

        For each block in State

            Check if the block can move then

                Check if the end state is not inside the royals then

                    Push the state to L[iy+1]

                    Have state parents of its before state and before state is included

                    Set the state's id to idCounter

                    Set discovered[idCounter] true

                    Increment idCounter

                    Add the edge of current state to found state to BFS

                    Check if the found state meets win condition then

                        Make won true

                        Break the loop

Add last element in royals to its parent vector

DFS:

```
dfs edge tree
state vector royalFamily to prevent cycles
state stack waitSide to push possible edges
push initial state to waitSide
bool array explored initiated false
win condition won set to false
idCounter set to 1
state dontWait to save popped states
while waitSide is not empty and won is false
    Take the top state from waitSide
    İf explored[top.id] is false then
        Set explored[top.id] to true
        Check if state top is not accured before then
            Add edge of its parent to it to bfs
            For each block in current state
                Check if it can move
                    Check if the next move is not accured before then
                        Add current state to waitSide
                        Add next state to waitSide
                        Set current state to next state's parent
                        Set next state's id to idCounter
                        Add next state to royalFamily
                        Increment idCounter
                        Check if win condition met then
                            Set won true
                            Break the loop
Add an edge consisting of waitSide's top state to itself to bfs
Add waitSide's top state to royalFamily
```

	BFS	DFS
Number of nodes generated	309	78
Maximum number of nodes kept in the memory	309	142
Running time	4.4e-5	8.7e-5

Explain your classes and your methods. What are their purposes?

There are 3 classes in my implementation: State, Block and Edge.

```
class Block //block class
{
public:
    Block(int a, int b, int d, char c)
    { x = a; y = b; direction = c; length = d; };
    Block() {};
    void setX(int a) { x = a; };
    void setY(int a) { y = a; };
    void setDirection(char a) { direction = a; };
    void setLength(int a) { length = a; };
    int getLength() { return length; };
    int getX() { return x; };
    int getY() { return y; };
    char getDirection() { return direction; };

private:
    int x;
    int y;
    int length;
    char direction;
};
```

Block consists of block’s properties and its get set methods.

```
class edge { //edge class
public:
    int u;
    int v;
    edge() {};
    edge(int a, int b) { u = a; v = b; };
};
```

Edge consists of u to v edge ids.

```cpp
class State //state class
{
public:
    vector<State> parents;
    vector<Block> blocks;
    int id;
    vector<vector<int > > fullMat;

    void matFill();
    void printState();
    bool isWon();
    State moveLeft(int order);
    State moveUp(int order);
    State moveRight(int order);
    State moveDown(int order);
    bool canLeft(int order);
    bool canRight(int order);
    bool canUp(int order);
    bool canDown(int order);
    bool noPath(State);
    State() {
    };
    State(int a) {
        id = a;
    };
    State operator=(State a) {   //
        this->parents = a.parents;
        this->blocks = a.blocks;
        this->fullMat = a.fullMat;
        this->id = a.id;
        return *this;
    };
private:
};
```

State is the class implemantation mainly carried out. It contains a game mat visualized in 2d vector, Block vector to keep track on the blocks it has on the mat, an id to specify each state when implementing bfs and dfs lastly, parents state vector to keep on track of the path for dfs and only one parent for the bfs implementation.

```cpp
bool State::canRight(int order) {
    if (blocks[order].getDirection() == 'h')
        return false;
    if (blocks[order].getX() != 5 && (fullMat[blocks[order].getX() + 1][blocks[order].getY()]) == 0)
    {
        return true;
    }
    else
        return false;
}
```

canDirection methods: to check if a block is moveable in sapecific direction.

```
State State::moveUp(int order) {
    State temp = *this;
    int label = temp.fullMat[temp.blocks[order].getX()][temp.blocks[order].getY()];
    temp.fullMat[temp.blocks[order].getX()][temp.blocks[order].getY()] = 0;
    temp.fullMat[temp.blocks[order].getX()][temp.blocks[order].getY() + temp.blocks[order].getLength()] = label;
    temp.blocks[order].setY(temp.blocks[order].getY() + 1);
    return temp;
}
```

moveDirection methods: to move the blocks and update their values.

```
bool State::isWon() {  //checks win condition
    bool yep = true;
    for (int a = blocks[0].getY() + blocks[0].getLength(); a < 6; a++) {
        if (fullMat[blocks[0].getX()][a] != 0) {
            yep = false;
            break;
        }
    }
    return yep;
}
```

İsWon: to check if the right side of the target block is empty and the win condition met.

```
bool State::noPath(State x) { //checks if the state is unique
                               //cycle check for dfs
    for (int i = 0; i < parents.size(); i++)
    {
        if (parents[i].fullMat == x.fullMat) {
            return false;
        }
    }
    return true;
}
bool noPath(vector<State>y, State x) {  //cycle check for bfs
    for (int i = 0; i < y.size(); i++)
    {
        if (y[i].fullMat == x.fullMat) {
            return false;
        }
    }
    return true;
}
```

noPath: to check if the given state was not accured before in current state's parents.

matFill: to fill the mat of the first state with respect to properties of its Block vector.

```
int maxMoveCalc(vector<Block> x) { //calc
    int max = 1;
    for (int i = 0; i < x.size(); i++)
    {
        max = max*(6 - x[i].getLength());
    }
    return max;
}
```

maxMoveCalc: to calculate maximum number of moves can be done on the game to initiate dynamic explored and discovered arrays.

```
void printPath(vector<State> arr) {  //prints states of the path on console as its requested in txt format
void printStatePath(vector<State> arr) { //prints mats of the states of given vector on console
void printToFile(string fileName, vector<State> arr) {//prints states of the path on file
void printToFileDFS(string fileName, vector<edge> arr, vector<State> arr2) { //prints states of the path on file
void printStatePathBFS(vector<edge> arr, vector<State> arr2) { //prints states of the path on console
void printStatePathDFS(vector<edge> arr, vector<State> arr2) { //prints states of the path on console
```

There are some print functions implemented as well.

2)

What is the extra complexity that is caused by the cycle search?

```
bool State::noPath(State x) { //checks if the state is unique
                                 //cycle check for dfs
    for (int i = 0; i < parents.size(); i++)
    {
        if (parents[i].fullMat == x.fullMat) {
            return false;
        }
    }
    return true;
}
bool noPath(vector<State>y, State x) {  //cycle check for bfs
    for (int i = 0; i < y.size(); i++)
    {
        if (y[i].fullMat == x.fullMat) {
            return false;
        }
    }
    return true;
}
```

My cycle check functions runs everytime a node to be added to bfs or dfs tree, scannning all the before unique states. If any matches it breaks the loop, other wise goes through all of it.

Leaving us an estimation of $\Theta(n*n)$, n search for nth node.

If you use adjacency list representation, how does the complexity of your algorithm change?

It would be reduces to $\Theta(m+n)$ complexity for both traversing implementations.

This course's Graph slide is used on preparing of this report and the implementation of the codes of the program.