University of West Bohemia

Faculty of Applied Sciences

Artificial Intelligence

Task 1: 8-puzzle

Rafał Karwowski

22.10.2020

Implementation of the tree structure

If start state is not the same as the goal state, it starts creating tree. To move deeper in the tree it needs a few information from the previous state.

```
zeroPosition = findZero(states, MinF);
g = states[MinF][3][0]+1;
previousMove = states[MinF][4][0];
```

First of all, it need to know the position of zero (***zeroPosition***) in matrices, this information is used to check in which direction move can be made. Information is taken from previous state with the lowest F(x) value, to know from which exactly it uses variable ***MinF*** (we will come back to how it is obtained later), and call method ***findZero***.

```
zeroPosition = findZero(states, MinF);
```

Because it creates deeper level it the tree, it have to take ***g*** value from the ***MinF*** state and adds 1.

```
g = states[MinF][3][0]+1;
```

To prevent moving back and forth, it checks the previous move of the ***MinF*** state and puts it to variable ***previousMove***.

```
previousMove = states[MinF][4][0];
```

Now, when it has all of this information, it starts expansion. The possible moves are: up, down, left and right. It has to go through every of them separately.

```
if(isMovePossible("up", zeroPosition) && previousMove != 2)
{

    for(int i=0; i<3; i++)
    {
        for(int j=0; j<3; j++)
        {
            if(zeroPosition[0] == i && zeroPosition[1] == j)
            {
                states[nr][i][j] = states[MinF][i-1][j];
            }else if(zeroPosition[0]-1 == i && zeroPosition[1] == j)
            {
                states[nr][i][j] = states[MinF][i+1][j];
            }else
            {
                states[nr][i][j] = states[MinF][i][j];
            }
        }
    }

    states[nr][3][0] = g;
    states[nr][3][1] = heuristicStates(states, nr)*importanceH; // h
    states[nr][3][2] = states[nr][3][0] + states[nr][3][1]; // f
    states[nr][4][0] = 1; //in which direction move

    nr++;

}
```

Firstly, it checks if move is possible. To do that it calls method ***isMovePossible*** and transmits two parameters: direction in which move is intend to be made and position of zero in the matrices. The second thing which has to be check is the ***previousMove***. E.g. if ***previousMove*** is equal to 2 (down), it does not make sense to create new state by moving up.

```
if(isMovePossible("up", zeroPosition) && previousMove != 2)
{
```

When the conditions are fulfilled, it creates new state by moving every element to the same place like before, except zero and its new position. They are swapped.

```
for(int i=0; i<3; i++)
{
    for(int j=0; j<3; j++)
    {
        if(zeroPosition[0] == i && zeroPosition[1] == j)
        {
            states[nr][i][j] = states[MinF][i-1][j];
        }else if(zeroPosition[0]-1 == i && zeroPosition[1] == j)
        {
            states[nr][i][j] = states[MinF][i+1][j];
        }else
        {
            states[nr][i][j] = states[MinF][i][j];
        }
    }
}
```

Now it attaches information about: ***g***, ***h***, ***f*** and the direction in which move was made, to the new state.

```
states[nr][3][0] = g;
states[nr][3][1] = heuristicStates(states, nr)*importanceH; // h
states[nr][3][2] = states[nr][3][0] + states[nr][3][1]; // f
states[nr][4][0] = 1; //in which direction move
```

The only thing left is to increase number of generated states and the program checks another position in which it can move.

```
    nr++;

}
```

When every possible move, from ***MinF*** state, is generated, it is time to find new ***MinF***.

```
while(!changed)
{
    for(int i=2; i < nr; i++)
    {
        if(states[i][3][2] <= f && states[i][4][1] != 1)
        {
            MinF = i;
            f = states[i][3][2];
            changed = true;

            if(compareStates(states, 0, MinF, 2))
            {
                done=true;
                break;
            }

        }
    }

    f++;
}
```

It looks for new state; therefore loop is active until it finds it.

```
while(!changed)
{
```

To find the lowest ***MinF*** it has to go through every state. Starting from ***i*** = 2, because ***i*** = 0 is the goal state and ***i*** = 1 is the start state.

```
for(int i=2; i < nr; i++)
{
```

For every state it checks if the ***f*** value is lower than the current one. To prevent being stuck in only one state or using over and over the same ones, it also checks if state has been used previously (***states[i][4][1]*** = 1 means that it has been used).

```
if(states[i][3][2] <= f && states[i][4][1] != 1)
{
```

When it finds state with lower ***f*** it saves its position, its ***f*** value and marks that the change has been made.

```
MinF = i;
f = states[i][3][2];
changed = true;
```

It is also time to check if the new lowest state is the goal state. If it is, the program marks it and moves out from loop, which generates states, to the rest of the program. If it is not, program continues.

```
if(compareStates(states, 0, MinF, 2))
{
    done=true;
    break;
}
```

Sometimes it does not find state with lower ***f*** than current one, so the ***f*** has to be incremented and the operation of finding lower ***f*** starts again.

```
    f++;
}
```

In the end of the loop which generates tree, informations, for next branch, have to be saved.

```
f--;
states[MinF][4][1] = 1;
correctMinF[howManyMinF] = MinF;

howManyMinF++;
```

Decrease ***f***, because it was increase one too many times.

```
f--;
```

Mark that the new state with the lowest ***f*** has been used.

```
states[MinF][4][1] = 1;
```

Save the new ***MinF*** state in different array. It is use in the next part of the program to find correct path, to solve the puzzle.

```
correctMinF[howManyMinF] = MinF;
```

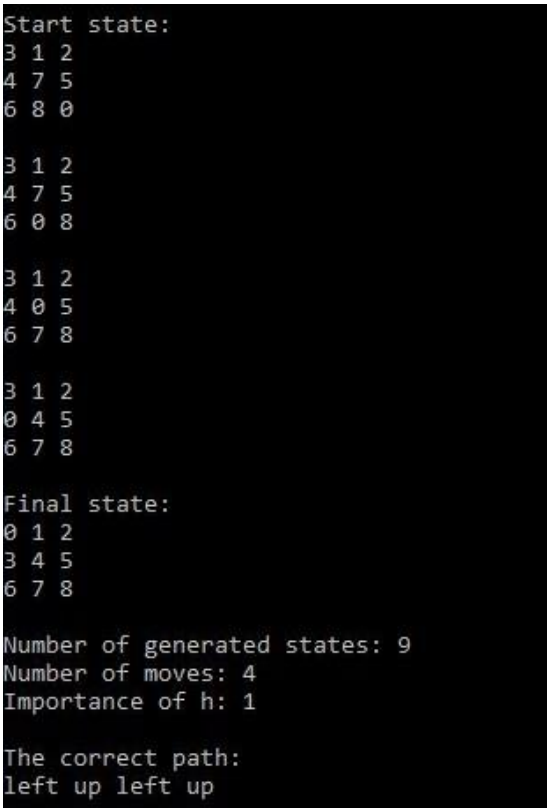
And add 1 to ***howManyMinF***.

```
howManyMinF++;
```

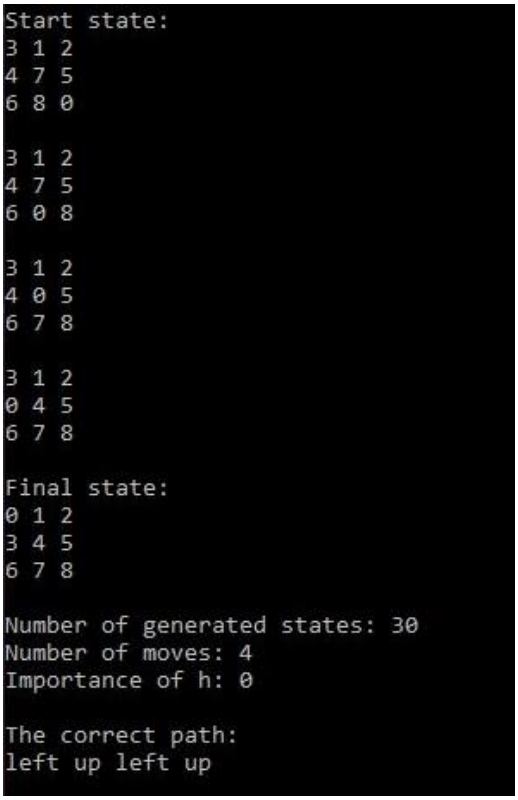
If ***MinF*** state is exactly the same like goal state the loop ends and program moves to another part. If not, the loop continues.

Results for h = 0

When ***h*** = number of misplaced tiles.

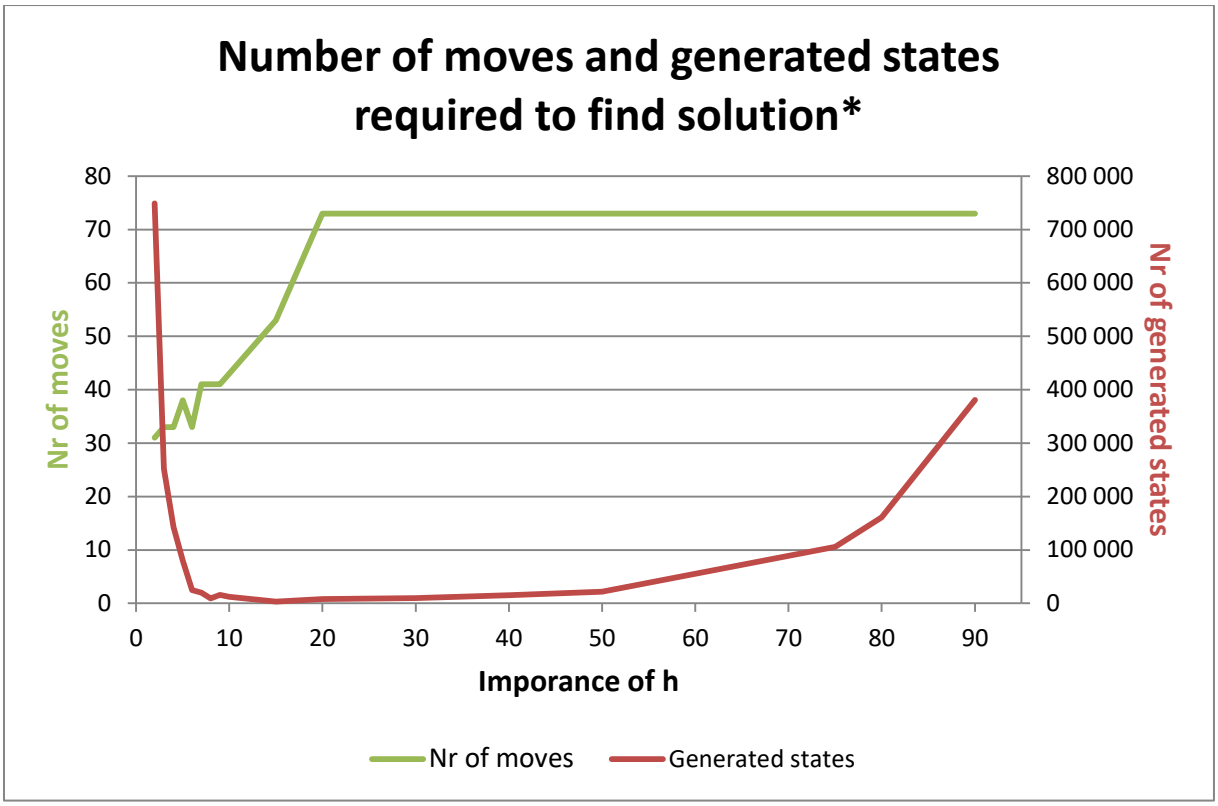


When program sets ***h***=0 for every state.



The second solution uses Breadth-First Search tree. It needs to generate more states, because it goes level by level, checking every possible move. But it guarantees that the solution is optimal.

The first solution uses A* algorithm. It focuses on importance of number of misplaced tiles. If implemented correctly, it can dramatically decrease number of generated states. Resulting in deceasing time and computer power required to solve task. But when implemented incorrectly, it can do the exact opposite. Another disadvantage of this approach is that the higher importance of ***h***, the larger the number of moves required to finding solution. The easiest way to observe it is by changing the importance of value ***h***.



*for this start state:

```
{8, 0, 6},
{5, 4, 7},
{2, 3, 1}
```