Coursework - Blocksworld Tile Puzzle

-COMP2208 Intelligent Systems-

Cristian Ioan Niculae (cin1g15)

Part A. Approach

My solution for the 'Blocksworld tile puzzle' was made entirely in C++. To solve this problem, I created 3 classes:

- Main class: for reading the initial state and the options for the tree search;
- State class: used to describe the nodes (states) of the tree. Functions found here (not including setters and getters):
  - **'std::string hash()'** hash function used for graph searching (extension). Returns a specific string for that state, which is used with a map, to determine if a **'State'** was visited before.
  - **'bool check()'** returns true if the state is a solution or false if it isn't.
  - **'void print()'** nicely prints the state: '0' for empty cell, '#' for agent, and 'A-z' for blocks.
  - **'void computeImp()'** computes the Manhattan distance between blocks of the current state and the closest solution (I consider every tower that respects the rules a solution to this puzzle)
  - **'void destroy()'** destroys the state (frees all the memory allocated by the state). A must have function in this problem, as it is memory demanding.
- TreeSearcher class: used for searching inside the tree. Functions found here:
  - **'void expand(State)'** function that expands nodes.
  - **'void startBFS()'** starts the breadth-first search on the tree.
  - **'bool startDFS(int)'** starts the depth-first search on the tree. (used also for iterative deepening)
  - **'void startAStar()'** starts the A* search on the tree.

Puzzle Solutions: As mentioned earlier, I consider every tower a solution of the puzzle if it respects the two rules of the puzzle:

- All of the blocks have to be one over each other.
- All of the blocks have to be in order: first row of the tower - A; second row of the tower - B; etc.

Expansion Order: The branching order can be either random or fixed, depending on the option selected at the beginning of the C++ program. We will see how the algorithms will behave with different expansion orders.
Note: If the branching order is random, BFS can give different results if there are multiple 'best' solutions (smallest level BFS can find a solution, has actually multiple solutions).

Memory limit: To test the full potential of all searching algorithms, I gave my program a stack limit of **12GB**.

Tests difficulty: To increase the test difficulty I increased the grid's size and increased the number of blocks: there are always **size**-1 blocks (where **size** denotes the size of the grid: if we have a 7x7 grid, we will have 6 blocks to arrange)

Graph Search vs Tree Search (extension): For this exercise I included Graph Search (I check if a node has been visited before) to show and compare it with Tree Search.

Reversing moves (extension): Another way to improve the Tree Search is to don't reverse the last move, as for example: Node A -up-> Node B -down-> Node C will always result in Node A = Node C. I show the difference of the implemented search algorithms with and without reversing the last move.

Obstacles (extension): Compare how implemented search methods perform when we include obstacles.

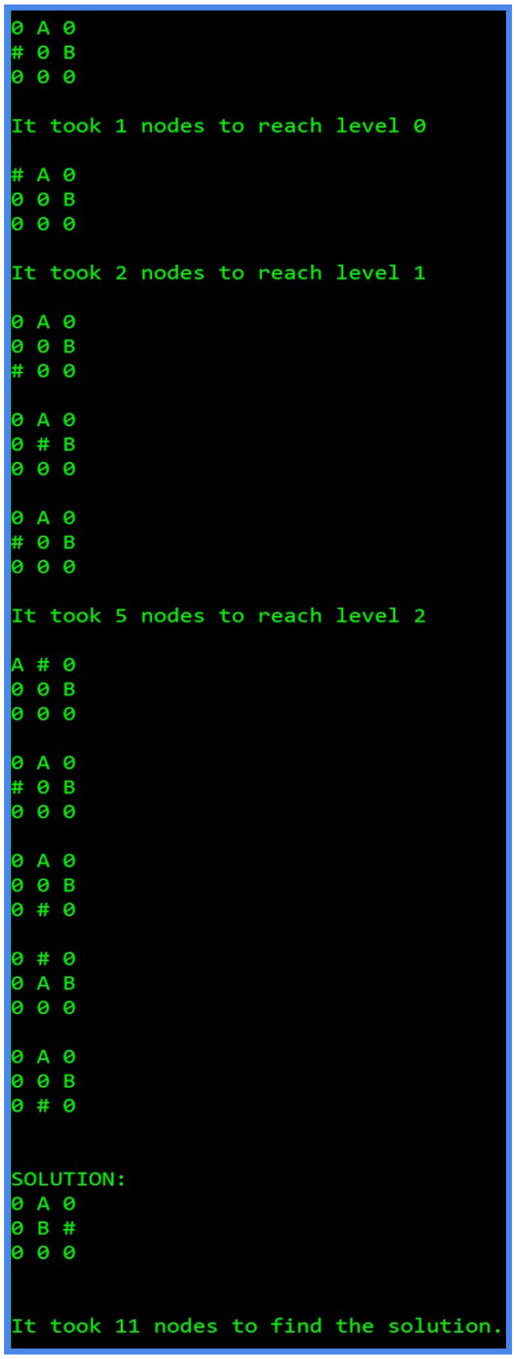
Part B.

Breadth-first search (BFS):

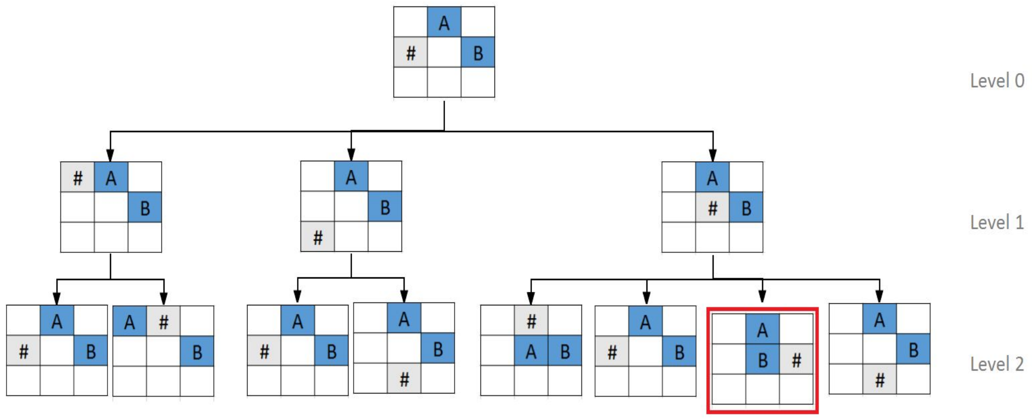
The logic behind BFS is simple: I use a queue (First in first out), which at the beginning contains only the starting node, and then do the following steps:

- 1) Take the top element of the queue and put it in a variable 'current'.
- 2) Check if 'current' is solution.
    - 2a) If yes then print the solution and end the BFS.
    - 2b) If no, extend the 'current' node, adding its children nodes to the queue, and repeating steps 1) and 2).

To show how BFS works I've chosen a very simple 3x3 puzzle, with only 2 blocks A and B (see the tree pictured below). The image on the right is a picture of the console with some debugging output implemented to show the nodes that BFS goes through to find the solution of the puzzle mentioned. As you can see, BFS starts with the root of the tree, and then expands all of its children. Before going to the next level, BFS will always expand all nodes on the current level; as seen in the right image: the first level 2 expanded node is the 5th one, meaning that to reach level 2 it had to expand all 3 nodes on level 1 and the root on level 0.



```
0 A 0
# 0 B
0 0 0

It took 1 nodes to reach level 0

# A 0
0 0 B
0 0 0

It took 2 nodes to reach level 1

0 A 0
0 0 B
# 0 0

0 A 0
0 # B
0 0 0

0 A 0
# 0 B
0 0 0

It took 5 nodes to reach level 2

A # 0
0 0 B
0 0 0

0 A 0
# 0 B
0 0 0

0 A 0
0 0 B
0 # 0

0 # 0
0 A B
0 0 0

0 A 0
0 0 B
0 # 0


SOLUTION:
0 A 0
0 B #
0 0 0


It took 11 nodes to find the solution.
```



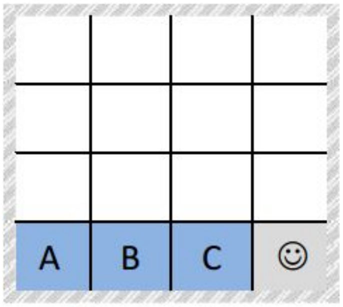
Depth-first search (DFS)

For DFS I use a stack, which at the beginning has only one element, the root of the tree. The algorithm can be described as follows:

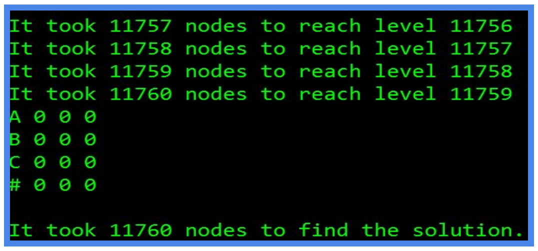
- 1) Take the top element of the stack and put it in a variable 'current'.
- 2) Check if 'current' is solution.

    - 2a) If yes then print the solution and end the DFS.
    - 2b) If no, extend the 'current' node, adding its children nodes to the queue (the order we put the children on the stack is random), and repeating steps 1) and 2).

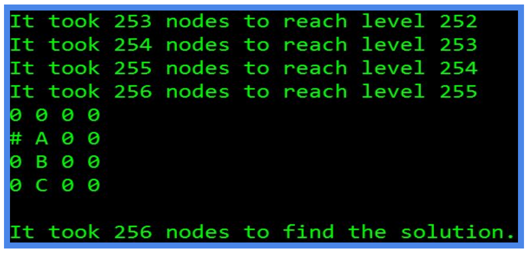
We said at **2b)** that the order we put the children on the stack is random because if we don't randomize the order of the children, we will end up making always the same move; e.g. going always 'up'. Because of this randomness, we don't know when we will find a solution. Let's take the example from the problem statement (image on right) and run DFS multiple times to compare the depth of the solution found:



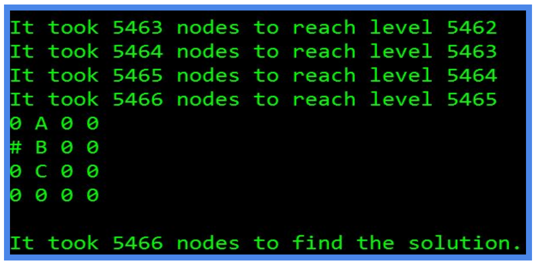
Start state.



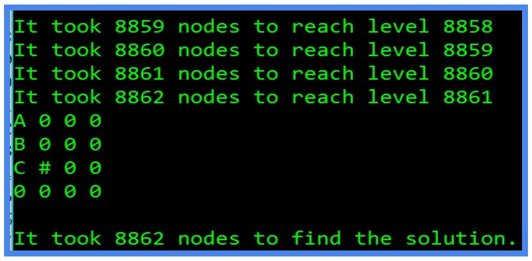
```
It took 11757 nodes to reach level 11756
It took 11758 nodes to reach level 11757
It took 11759 nodes to reach level 11758
It took 11760 nodes to reach level 11759
A 0 0 0
B 0 0 0
C 0 0 0
# 0 0 0

It took 11760 nodes to find the solution.
```



```
It took 253 nodes to reach level 252
It took 254 nodes to reach level 253
It took 255 nodes to reach level 254
It took 256 nodes to reach level 255
0 0 0 0
# A 0 0
0 B 0 0
0 C 0 0

It took 256 nodes to find the solution.
```



```
It took 5463 nodes to reach level 5462
It took 5464 nodes to reach level 5463
It took 5465 nodes to reach level 5464
It took 5466 nodes to reach level 5465
0 A 0 0
# B 0 0
0 C 0 0
0 0 0 0

It took 5466 nodes to find the solution.
```



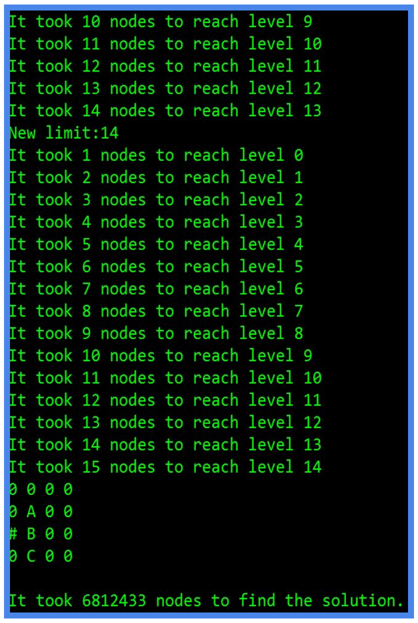
```
It took 8859 nodes to reach level 8858
It took 8860 nodes to reach level 8859
It took 8861 nodes to reach level 8860
It took 8862 nodes to reach level 8861
A 0 0 0
B 0 0 0
C # 0 0
0 0 0 0

It took 8862 nodes to find the solution.
```

<u>Note</u>: As stated in the beginning, I define a solution as a node that respects the two rules of the puzzle, so as you can see, same puzzle can have different solutions.

Iterative Deepening (IDS)

For IDS I used the same function as for DFS but with a limit, as follows:

- 1) Have a variable 'limit', which at the beginning is initialized with 1;
- 2) Use the DFS function with the function's option to limit the search until 'limit':
    - If the current state's level is equal or bigger than 'limit', then we just pop it from the stack, without expanding it;
- 3) If we didn't find a solution in the previous step, increment 'limit' with one and repeat step 2);



```
It took 10 nodes to reach level 9
It took 11 nodes to reach level 10
It took 12 nodes to reach level 11
It took 13 nodes to reach level 12
It took 14 nodes to reach level 13
New limit:14
It took 1 nodes to reach level 0
It took 2 nodes to reach level 1
It took 3 nodes to reach level 2
It took 4 nodes to reach level 3
It took 5 nodes to reach level 4
It took 6 nodes to reach level 5
It took 7 nodes to reach level 6
It took 8 nodes to reach level 7
It took 9 nodes to reach level 8
It took 10 nodes to reach level 9
It took 11 nodes to reach level 10
It took 12 nodes to reach level 11
It took 13 nodes to reach level 12
It took 14 nodes to reach level 13
It took 15 nodes to reach level 14
0 0 0 0
0 A 0 0
# B 0 0
0 C 0 0

It took 6812433 nodes to find the solution.
```

Again, because we use DFS to solve IDS, there will be that randomize of the children nodes, so we aren't guaranteed that we will always get the same solution, but this time, we are guaranteed that we will always get a best solution (there can be multiple best solution so we can't be sure which one of those we will get).

The image on the left is IDS checked against the same puzzle as earlier (example from problem statement). We can see that the solution found is actually one of the best solution (depth 14), but it takes "6812433" nodes to find it, which is a lot bigger than the largest number of expanded nodes from the DFS tests (out of 4 tests the largest was 11760)

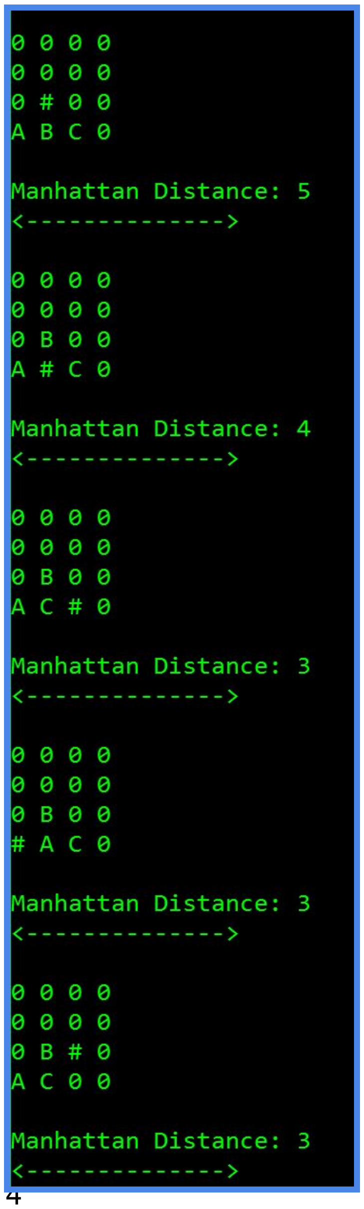
<u>Note</u>: When it says that "It took x nodes to reach level y" is it just an interior node counter of the current DFS, to prove that to solve IDS I used the

DFS function. The total nodes counter can be seen at the end: "It took 6812433 nodes to find the solution".

A* Search

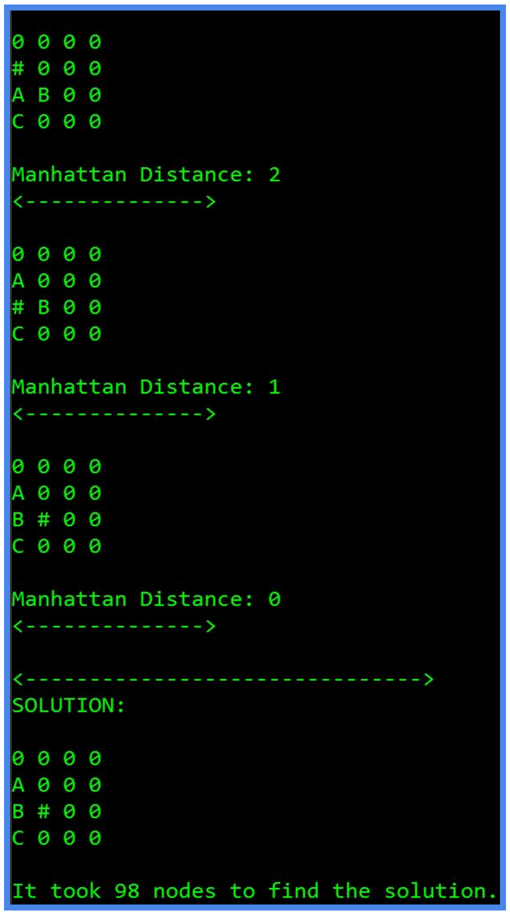
A* Search is an informed search algorithm that uses heuristics to find the solution faster (with a number of expanded nodes lower) than previous search algorithms. For this exercise I used as an heuristic function the Manhattan distance of the blocks to their final target (as I have multiple solution I also have multiple Manhattan distances, but I always take the lowest one). For this algorithm I use a priority queue, initialized with the startState, and in which I use the Manhattan distances of two nodes to compare them, so the nodes with lowest Manhattan distance will always be on top. This algorithm can be described using the following steps:

- 1) Take the top of the priority queue and putting it in a variable 'current'.
- 2) Check if 'current' node is solution (or if it has Manhattan distance equal to 0):
    - 2a) If yes, then output solution and return.
    - 2b) If not, then expand the 'current' node and before adding the children nodes of 'current' to the priority queue computer the heuristic function (I store the result of heuristic function inside a variable in the class State, so that I won't have to compute it multiple times). Repeat from step 1).



```
0 0 0 0
0 0 0 0
0 # 0 0
A B C 0

Manhattan Distance: 5
<-------------->

0 0 0 0
0 0 0 0
0 B 0 0
A # C 0

Manhattan Distance: 4
<-------------->

0 0 0 0
0 0 0 0
0 B 0 0
A C # 0

Manhattan Distance: 3
<-------------->

0 0 0 0
0 0 0 0
0 B 0 0
# A C 0

Manhattan Distance: 3
<-------------->

0 0 0 0
0 0 0 0
0 B # 0
A C 0 0

Manhattan Distance: 3
<-------------->
```

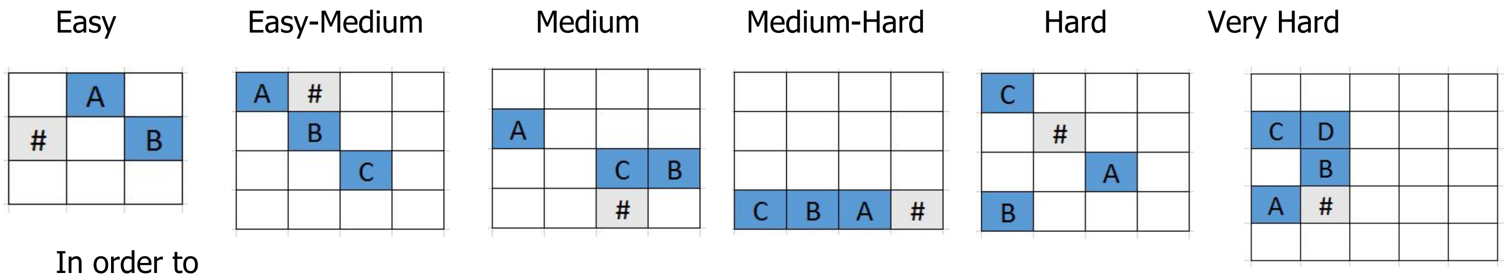
I implemented A* with the same randomness (from IDS and DFS) and by testing it against the previous puzzle (the one from the problem statement) you can see how well A* performs: only 98 nodes expanded to find a solution (because of the randomness that number can be larger or smaller, but not with a lot). The first image shows how the Manhattan distance behaves when used as an heuristic for A*.



```
0 0 0 0
# 0 0 0
A B 0 0
C 0 0 0

Manhattan Distance: 2
<-------------->

0 0 0 0
A 0 0 0
# B 0 0
C 0 0 0

Manhattan Distance: 1
<-------------->

0 0 0 0
A 0 0 0
B # 0 0
C 0 0 0

Manhattan Distance: 0
<-------------->

<------------------------------->
SOLUTION:

0 0 0 0
A 0 0 0
B # 0 0
C 0 0 0

It took 98 nodes to find the solution.
```

Part C. Scalability Study

To compare the 4 search methods I selected the following test cases and sorted them by difficulty:

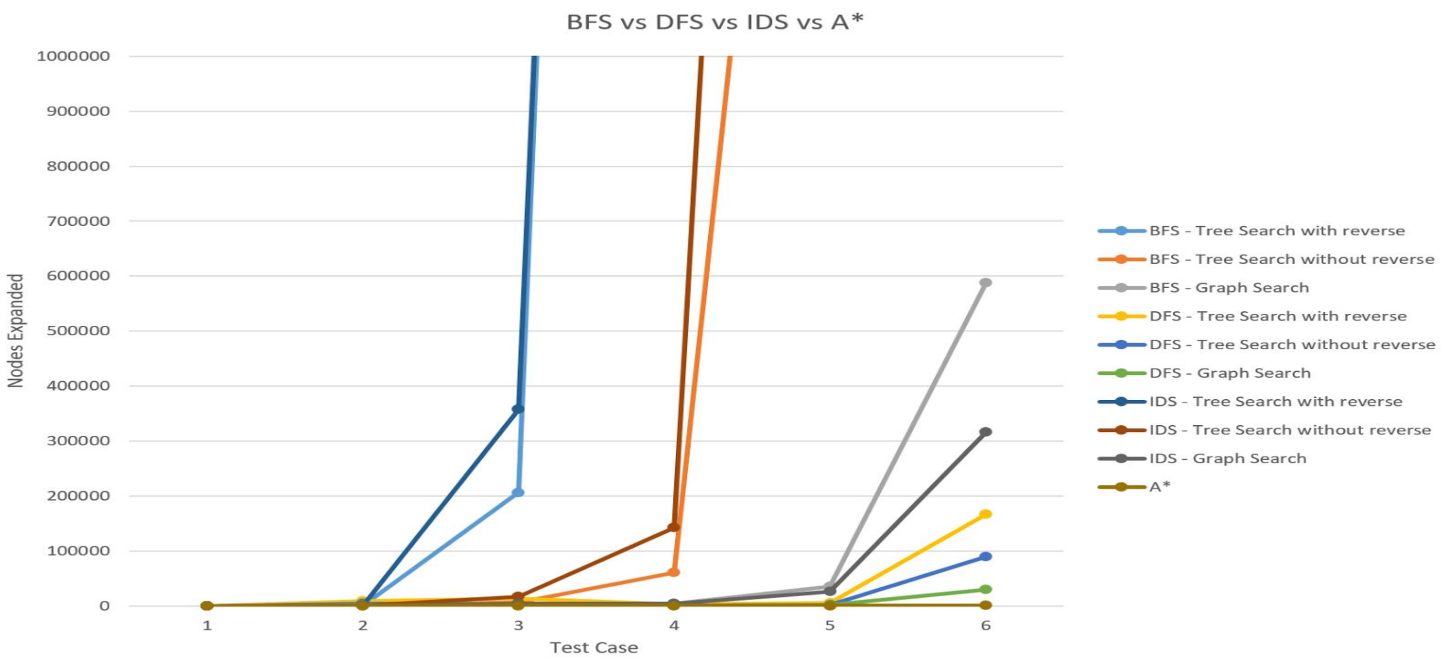


In order to increase the difficulty I had the following options:

- Change the blocks on the grid so the 'best solution' is at a bigger depth. (e.g. from 'Medium-Hard' to 'Hard')
- Add another block and make the grid bigger (by adding another block I have to add another row and column) and make sure that the best solution is not on a smaller depth than before. (e.g. from 'Hard' to 'Very Hard')

What I've seen is that most of my test cases are not passed by the requested searches, so I've done some improvements that are pointed out:

- **'without reverse'** = can't reverse the last movement (max branching factor of 3 instead of 4)
- **'graph search'** = I check if the current node has been visited before.



There are multiple things to be observed from the chart:

- Branching factor is very important; the difference between maximum branching factor of 4 and maximum branching factor of 3 is huge when we look at the difference between BFS and IDS with and without reverse.

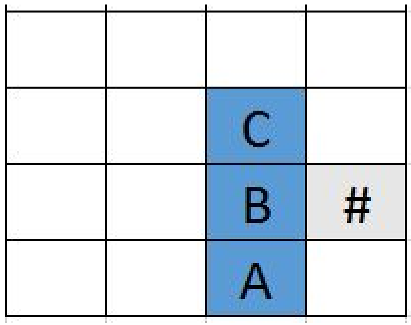
- A* with Manhattan distance has the best performance out of all implemented searches; even better than graph search (even though it has a big issue which we will talk about in the next part).
- Graph Search improves a lot! Knowing the states you visited before even allowed BFS to finish all the test cases.
- DFS has worst performance on 'Medium' test case than on 'Medium-Hard', probably because there aren't so many paths to a solution on that test case.

Part D. Extras and Limitations

Limitations:

Firstly, we are going to talk about limitations. The biggest limitation on this exercise was the memory limitation: to find a solution on 'Medium-Hard' test case with BFS (the version with 'Tree Search with reverse') I had to give my program a stack of 12GB, that's why I couldn't get any more results after 'Medium-Hard', so maybe if I would have taken a more memory friendly approach or if I had more resources I could had given at least one more result for BFS.

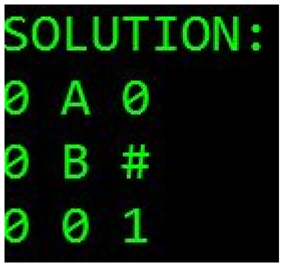
Another limitation was that for this particular problem, A* having Manhattan distance as heuristic function had big problems with some inputs; for example, look at the following input:



What's wrong with this input? Well, as you can see this input is not a solution to the puzzle (it should be A over B over C not C over B over A), and we can safely say that the Manhattan distance is bigger than 0 because as we just said it is not a solution. But if we use A* tree search on this input it will enter an infinite loop, as the agent will never change its place with a block; doing so, it will increase the Manhattan distance, so A* won't take that path (we have infinite of nodes with the same Manhattan distance as the input).

Extras:

- Puzzle with obstacles. Implemented the option to mark a wall (obstacle) in a puzzle. When printing the solution, the obstacle is represented by a '1' like in the image below.



```
SOLUTION:
0 A 0
0 B #
0 0 1
```

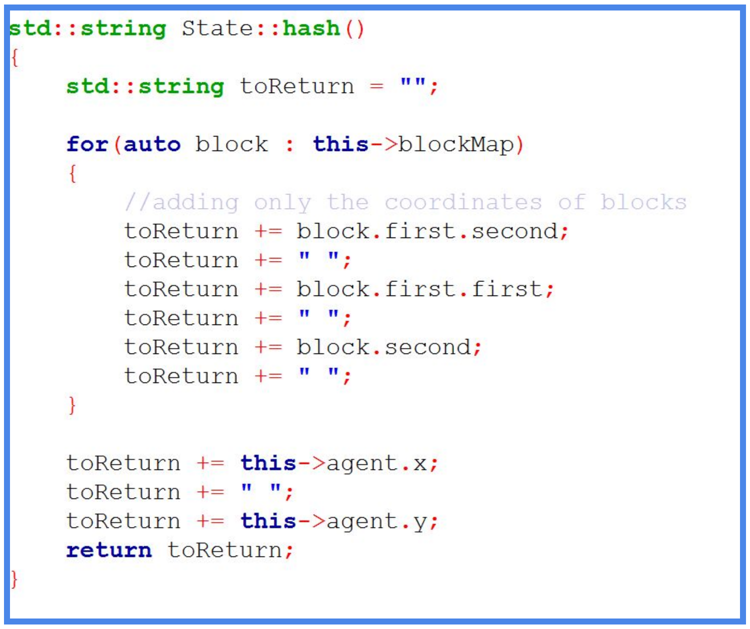
- Implemented Graph Search to compare it with Tree Search.
- Observed that the implementation of the puzzle can be highly improved by not expanding opposite to the last move (if the node's last move was 'up' we won't want to expand 'down' to get back where we started).

Part E. Code

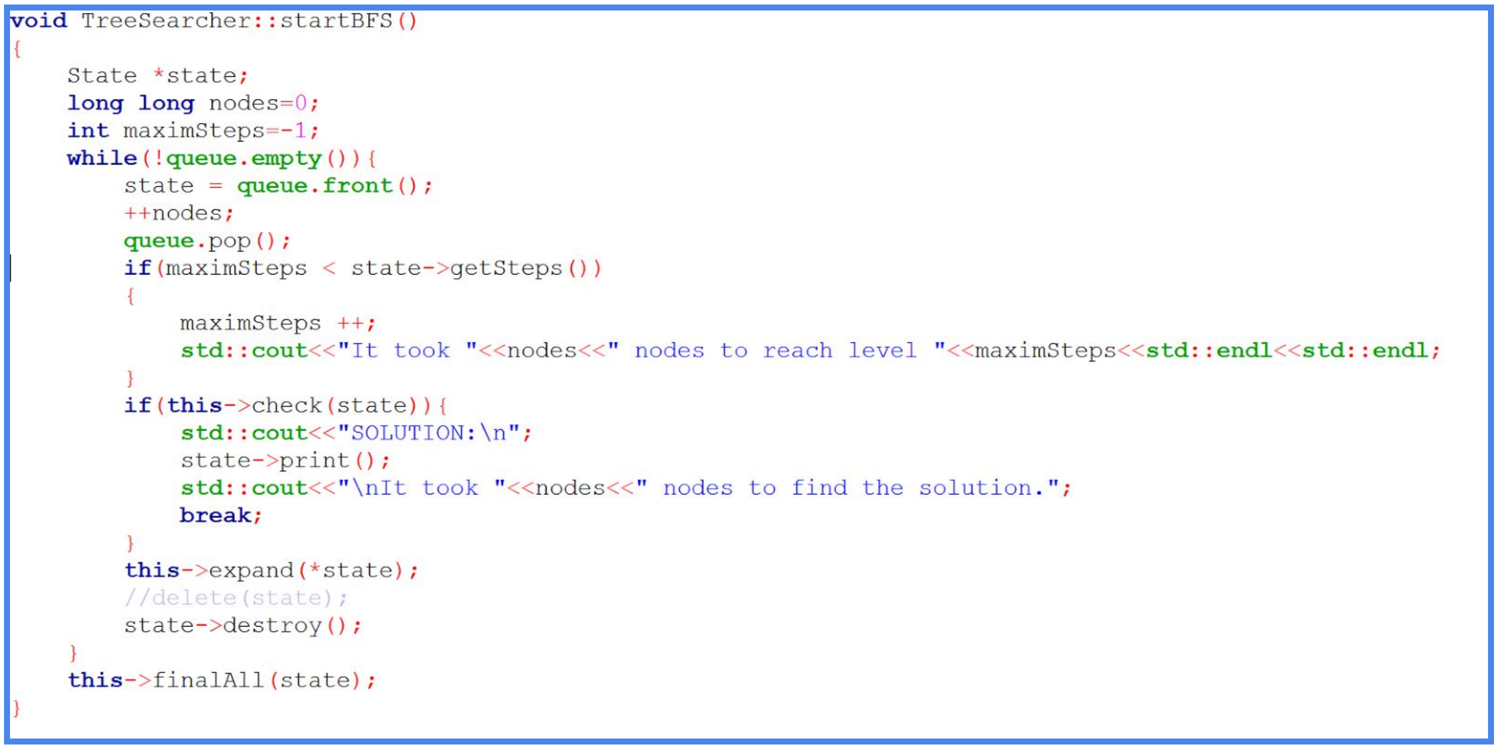
```
State* State::moveAgent(short y,short x)
{
    agent.x +=x;
    agent.y +=y;

    State *newState;

    //returning if agent is on an obstacle or out of bounds
    if(agent.x > size || agent.x < 1 || agent.y > size || agent.y < 1 || obstMap[std::make_pair(agent.y,agent.x)]==1)
    {
        agent.x -=x;
        agent.y -=y;
        newState = new State(&blockMap,&agent,this->size,this->steps,this,&obstMap);
        return newState;
    }else
    {
        //if we checked the last condition and it was false, it created a new element in obstMap that should be erased
        obstMap.erase(std::make_pair(agent.y,agent.x));
    }

    std::pair<short,short> p = std::make_pair(agent.y,agent.x);
    std::map<std::pair<short,short>, short>::iterator it =  blockMap.find(p);
    //this->printConfiguration();


    //creating the new State:
    if(it != blockMap.end())
    {
        //if the new state need to swap between agent and a block
        std::map<std::pair<short,short>, short> blockMap2 = blockMap;
        std::map<std::pair<short,short>, short>::iterator it2 = blockMap2.find(p);

        blockMap2.erase(it2);

        p.first -= y;
        p.second -= x;

        blockMap2[p]  = it->second;
        newState = new State(&blockMap2,&agent,size,this->steps+1,this,&obstMap);

    }else
    {
        //if the new state doesn't need to swap between agent and blocks
        std::map<std::pair<short,short>, short> blockMap2 = blockMap;
        newState = new State(&blockMap2,&agent,size,this->steps+1,this,&obstMap);
    }

    newState->setPreviousMove(y,x);

    //undoing the changes we have made on the agent
    agent.x -=x;
    agent.y -=y;

    return newState;
}
```

```
void State::computeImp()
{
    //First of all, creating a new map with the elements of the first map but inverted.
    for(auto bl : blockMap){
        invertedBlockMap[bl.second] = bl.first;
    }
    //initializing minimumDist with a very large number;
    int minimumDist=999999999,currentDist;
    for(int j = 1; j<=size; ++j)
    {
        currentDist=0;
        //computing the distance for solutions that begin on last row
        for(int i = size; i>1; --i)
        {
            currentDist+= abs(invertedBlockMap[i-1].first - i) + abs(invertedBlockMap[i-1].second - j);
            /*if(invertedBlockMap[i-1].second == j && invertedBlockMap[i-1].first != i)
            {
                currentDist++;
            }*/
        }
        //always selecting the minimum
        minimumDist = std::min(currentDist,minimumDist);

        currentDist=0;
        //computing the distance for solutions that begin on last-1 row
        for(int i = size-1; i>0; --i)
        {
            currentDist+= abs(invertedBlockMap[i].first - i) + abs(invertedBlockMap[i].second - j);
            /*if(invertedBlockMap[i].second == j && invertedBlockMap[i].first != i)
            {
                currentDist++;
            }*/
        }
        //always selecting the minimum
        minimumDist = std::min(currentDist,minimumDist);
    }

    this->imp = minimumDist;
}
```

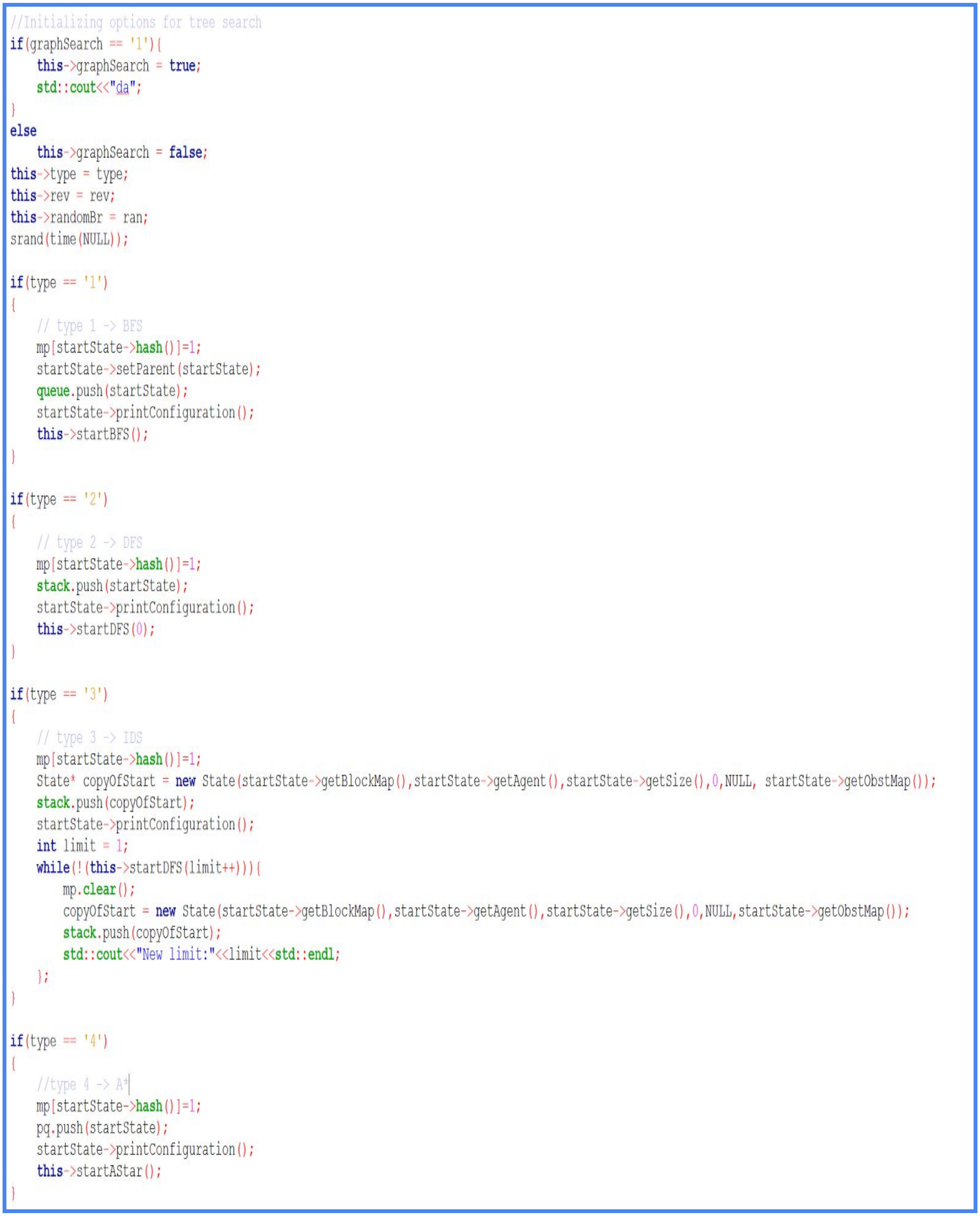
```
bool State::check()
{
    std::pair<short,short> p;

    //Finds the last block (e.g. if the size is 4, it finds the block 'C')
    for(auto block : this->blockMap)
    {
        if(block.second == this->size-1)
        {
            p = block.first;
            break;
        }
    }

    //if the last block is not on the last 2 rows, return false
    if(p.first < this->size - 1)
        return false;

    //returns true if everything is okay, or false if the order is not correct
    for(short i = 2; i < this->size; ++i)
    {
        p.first--;
        if(blockMap[p]!=this->size-i)
        {
            if(blockMap[p]==0)
                blockMap.erase(p);
            return false;
        }
    }

    return true;
}
```



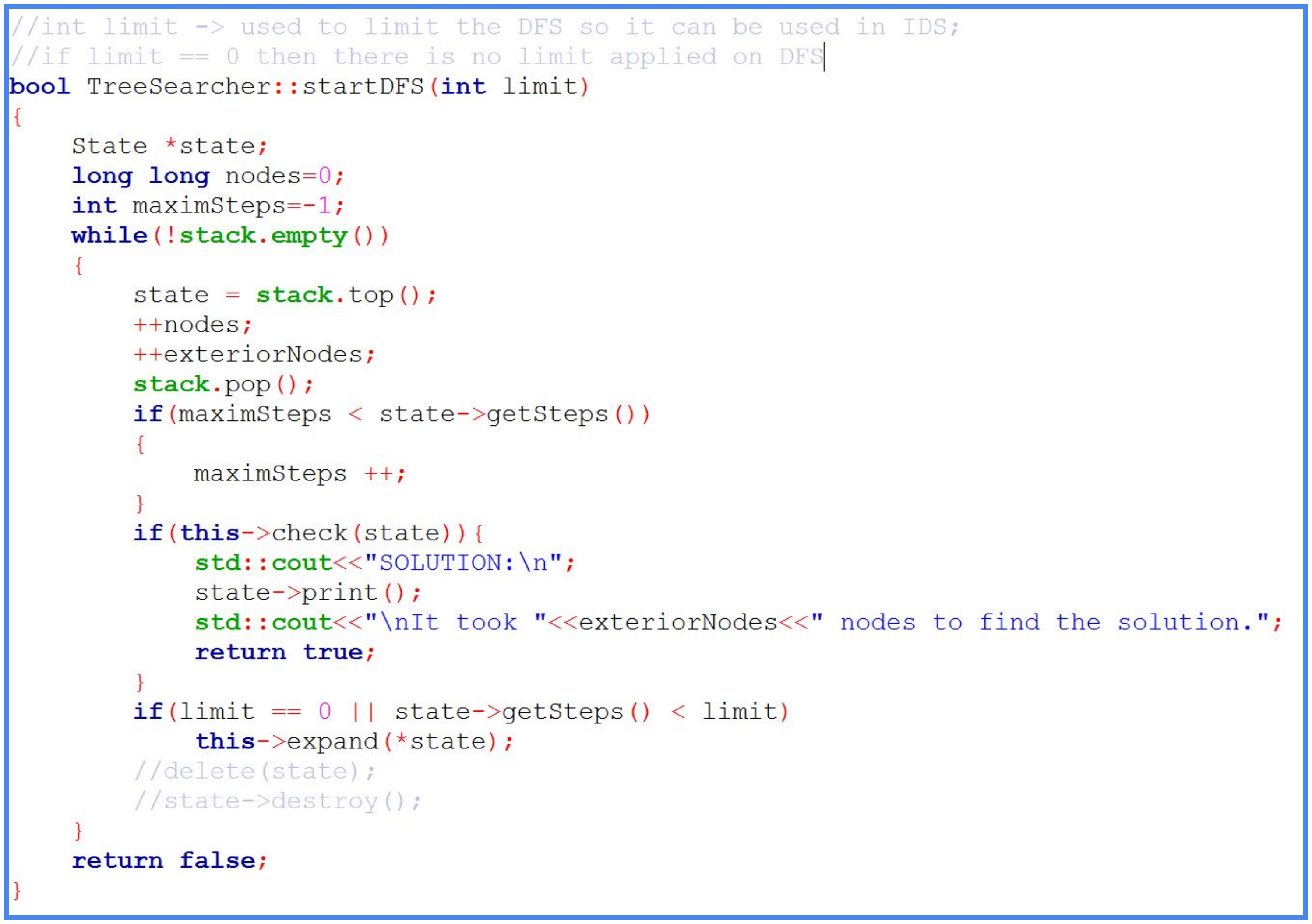
```
std::string State::hash()
{
    std::string toReturn = "";

    for(auto block : this->blockMap)
    {
        //adding only the coordinates of blocks
        toReturn += block.first.second;
        toReturn += " ";
        toReturn += block.first.first;
        toReturn += " ";
        toReturn += block.second;
        toReturn += " ";
    }

    toReturn += this->agent.x;
    toReturn += " ";
    toReturn += this->agent.y;
    return toReturn;
}
```



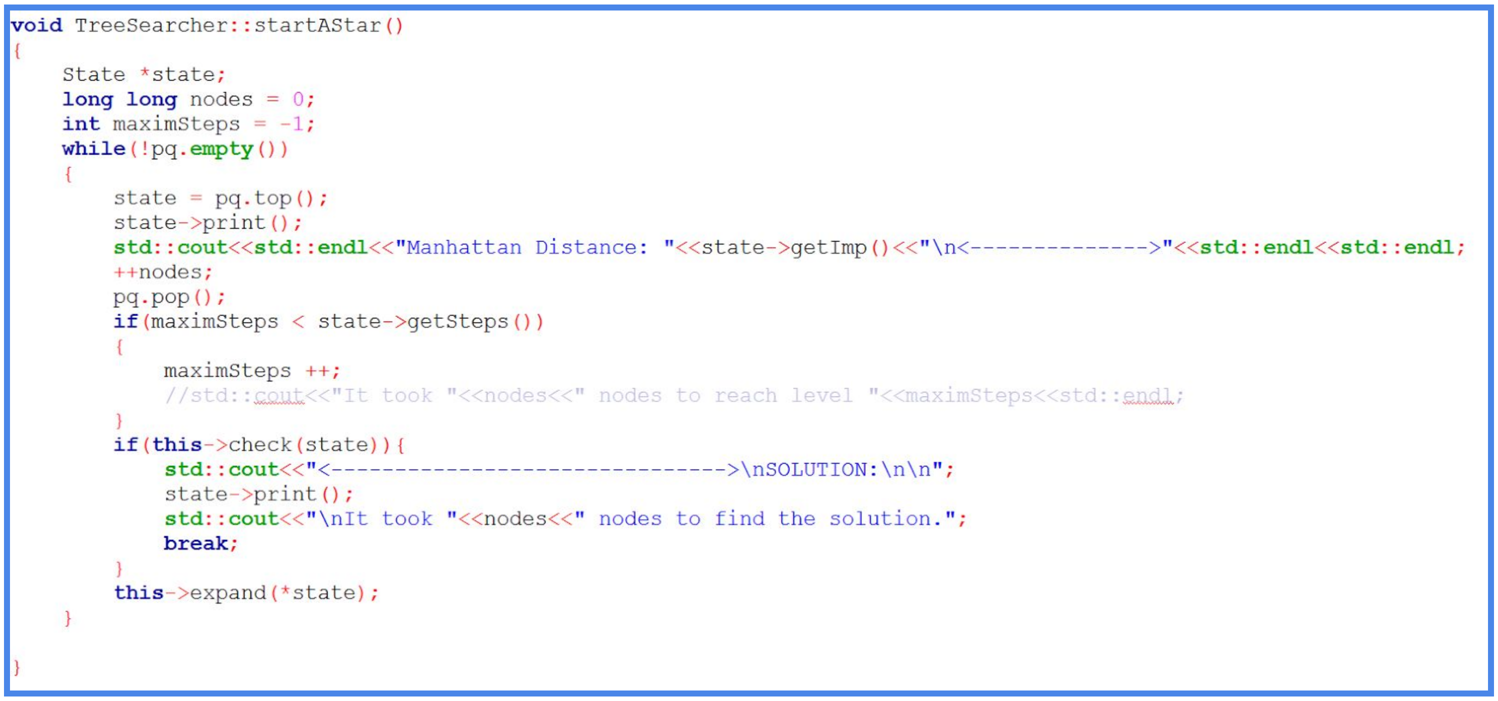
```
void TreeSearcher::startBFS()
{
    State *state;
    long long nodes=0;
    int maximSteps=-1;
    while(!queue.empty()){
        state = queue.front();
        ++nodes;
        queue.pop();
        if(maximSteps < state->getSteps())
        {
            maximSteps ++;
            std::cout<<"It took "<<nodes<<" nodes to reach level "<<maximSteps<<std::endl<<std::endl;
        }
        if(this->check(state)){
            std::cout<<"SOLUTION:\n";
            state->print();
            std::cout<<"\nIt took "<<nodes<<" nodes to find the solution.";
            break;
        }
        this->expand(*state);
        //delete(state);
        state->destroy();
    }
    this->finalAll(state);
}
```



```cpp
//Initializing options for tree search
if(graphSearch == '1'){
    this->graphSearch = true;
    std::cout<<"da";
}
else
    this->graphSearch = false;
this->type = type;
this->rev = rev;
this->randomBr = ran;
srand(time(NULL));

if(type == '1')
{
    // type 1 -> BFS
    mp[startState->hash()]=1;
    startState->setParent(startState);
    queue.push(startState);
    startState->printConfiguration();
    this->startBFS();
}

if(type == '2')
{
    // type 2 -> DFS
    mp[startState->hash()]=1;
    stack.push(startState);
    startState->printConfiguration();
    this->startDFS(0);
}

if(type == '3')
{
    // type 3 -> IDS
    mp[startState->hash()]=1;
    State* copyOfStart = new State(startState->getBlockMap(),startState->getAgent(),startState->getSize(),0,NULL, startState->getObstMap());
    stack.push(copyOfStart);
    startState->printConfiguration();
    int limit = 1;
    while(!(this->startDFS(limit++))){
        mp.clear();
        copyOfStart = new State(startState->getBlockMap(),startState->getAgent(),startState->getSize(),0,NULL,startState->getObstMap());
        stack.push(copyOfStart);
        std::cout<<"New limit:"<<limit<<std::endl;
    };
}

if(type == '4')
{
    //type 4 -> A*
    mp[startState->hash()]=1;
    pq.push(startState);
    startState->printConfiguration();
    this->startAStar();
}
```



```cpp
//int limit -> used to limit the DFS so it can be used in IDS;
//if limit == 0 then there is no limit applied on DFS
bool TreeSearcher::startDFS(int limit)
{
    State *state;
    long long nodes=0;
    int maximSteps=-1;
    while(!stack.empty())
    {
        state = stack.top();
        ++nodes;
        ++exteriorNodes;
        stack.pop();
        if(maximSteps < state->getSteps())
        {
            maximSteps ++;
        }
        if(this->check(state)){
            std::cout<<"SOLUTION:\n";
            state->print();
            std::cout<<"\nIt took "<<exteriorNodes<<" nodes to find the solution.";
            return true;
        }
        if(limit == 0 || state->getSteps() < limit)
            this->expand(*state);
        //delete(state);
        //state->destroy();
    }
    return false;
}
```



```cpp
void TreeSearcher::startAStar()
{
    State *state;
    long long nodes = 0;
    int maximSteps = -1;
    while(!pq.empty())
    {
        state = pq.top();
        state->print();
        std::cout<<std::endl<<"Manhattan Distance: "<<state->getImp()<<"\n<-------------->"<<std::endl<<std::endl;
        ++nodes;
        pq.pop();
        if(maximSteps < state->getSteps())
        {
            maximSteps ++;
            //std::cout<<"It took "<<nodes<<" nodes to reach level "<<maximSteps<<std::endl;
        }
        if(this->check(state)){
            std::cout<<"<------------------------------->\nSOLUTION:\n\n";
            state->print();
            std::cout<<"\nIt took "<<nodes<<" nodes to find the solution.";
            break;
        }
        this->expand(*state);
    }

}
```