Search Methods

COMP2208 Intelligent Systems

Student ID: 29338743

Metodi Istatkov

11/28/18

Approach

In my implementation I have created three different classes – Node, Searches, Main. My Node class is meant to represent each node in the trees that will be traversed by the four searching algorithms (BFS, DFS, IDS, A*). Every node is actually a grid, which shows the tile positions of the three blocks and the agent. Codewise I have implemented the grid as a matrix of chars so each block is represented by the chars 'A', 'B', 'C', and the agent is represented by the '@' character (as you will see in the next section). Every node in my implementation has a reference to its parent and also stores all of its children. The purpose of that is to allow recreation of the path to the solution. I am using the graph search version of BFS to find the shortest path to the solution. Then once BFS has discovered the final state, I am backtracking and finding the path to the solution. By having the path I can then control problem difficulty (more about this in the scalability study section).

Since the tree search version of DFS is very likely to enter an infinite loop and thus never end I am randomising the order in which the next move is selected. Also I am implementing DFS using a stack, because DFS works by expanding the last visited node so a LIFO data structure is quite suitable. There is also the option for recursive implementation, but I decided that an iterative approach would be more efficient.

For BFS I am using a queue data structure, because it checks the whole level before moving to the next one, so all children nodes will be checked only after all nodes from the parent level have been polled from the queue.

Iterative deepening depth first search is a hybrid between DFS and BFS. I am implementing it using a stack, but I am adding new nodes to the stack only if they are within the allowed depth limit. Once the depth limit is reached and the stack becomes empty, the root node is pushed and the search starts again from the beginning, and also the depth limit is increased by 1.

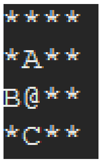
In terms of the A* searching algorithm I decided to use Manhattan distance as my heuristics instead of counting the number of misplaced tiles because it becomes slightly more efficient as the size of the problem increases.

Evidence

As evidence that the methods work I have made my code to print the configuration for each node. In order for the evidences to be easy to follow and understand, I will be changing the start state to being only one step away from the solution. The order in which the movements are made for expanding nodes is: right, left, up, down (when possible to make all).

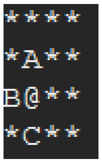
BFS: (Nodes added to the queue, left to right)

Root added to queue:



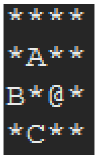
```
****
*A**
B@**
*C**
```

Removed node:

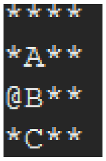


```
****
*A**
B@**
*C**
```

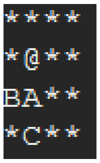
Expanded nodes from currently checked node:



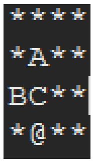
```
****
*A**
B*@*
*C**
```



```
****
*A**
@B**
*C**
```

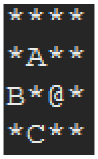


```
****
*@**
BA**
*C**
```



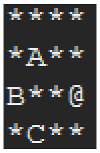
```
****
*A**
BC**
*@**
```

Removed node:

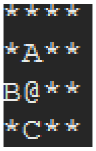


```
****
*A**
B*@*
*C**
```

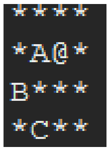
Expanded nodes from currently checked node:



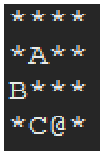
```
****
*A**
B**@
*C**
```



```
****
*A**
B@**
*C**
```

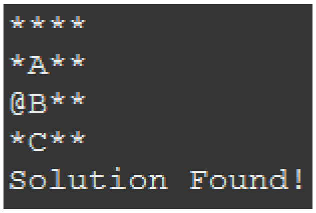


```
****
*A@*
B***
*C**
```



```
****
*A**
B***
*C@*
```

Removed node:

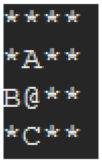


```
****
*A**
@B**
*C**
Solution Found!
```

The solution is the second expanded node from the start state and it is reached after the node that was entered before it on the queue is removed.

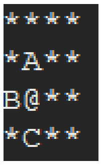
IDS: (Nodes added to the stack up to the depth limit, left to right)

Root added to stack:



```
****
*A**
B@**
*C**
```

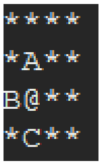
Popped from stack Depth Limit = 0 Nodes in the Stack:



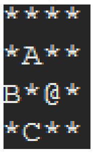
```
****
*A**
B@**
*C**
```

None, so stack becomes empty and the root is pushed and the depth limit is increased by 1(starting over again).

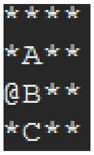
Popped from stack Depth Limit = 1 Nodes in the Stack:



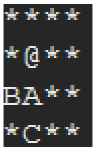
```
****
*A**
B@**
*C**
```



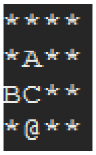
```
****
*A**
B*@*
*C**
```



```
****
*A**
@B**
*C**
```

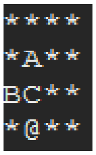


```
****
*@**
BA**
*C**
```

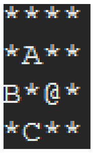


```
****
*A**
BC**
*@**
```

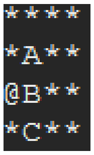
Popped from stack Depth Limit = 1 Nodes in the Stack:



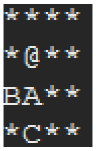
```
****
*A**
BC**
*@**
```



```
****
*A**
B*@*
*C**
```

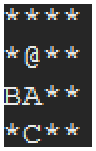


```
****
*A**
@B**
*C**
```

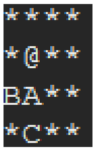


```
****
*@**
BA**
*C**
```

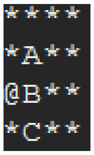
Popped from stack Depth Limit = 1 Nodes in the Stack:



```
****
*@**
BA**
*C**
```

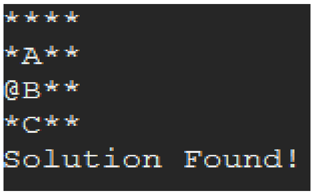


```
****
*@**
BA**
*C**
```



```
****
*A**
@B**
*C**
```

Popped from stack Depth Limit = 1

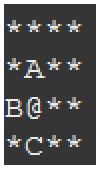


```
****
*A**
@B**
*C**
Solution Found!
```

It can be seen that the next checked node (popped from stack) is the node that was added last. Until the depth limit is reached nodes are added to the stack in the same fashion as DFS, then when going backwards they are popped one by one, again in DFS fashion.

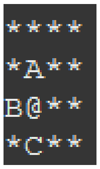
A*: (Nodes added to the priority queue and ordered based on their heuristic)

Root added to priority queue:



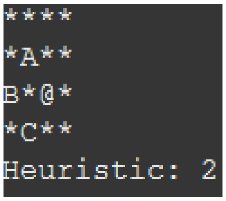
```
****
*A**
B@**
*C**
```

Removed node:

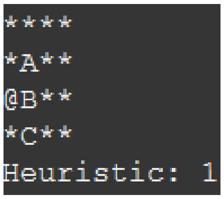


```
****
*A**
B@**
*C**
```

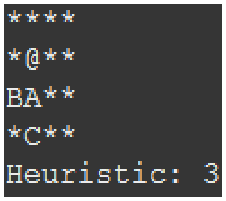
Expanded nodes from currently checked node:



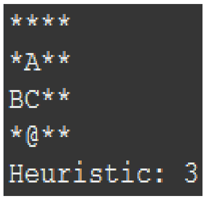
```
****
*A**
B*@*
*C**
Heuristic: 2
```



```
****
*A**
@B**
*C**
Heuristic: 1
```

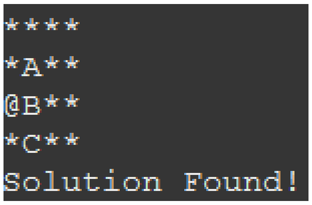


```
****
*@**
BA**
*C**
Heuristic: 3
```



```
****
*A**
BC**
*@**
Heuristic: 3
```

Removed node:

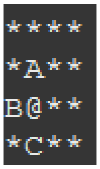


```
****
*A**
@B**
*C**
Solution Found!
```

No matter of the order in which the nodes are added to the priority queue, by seeing the next expanded node it is evident that the head of the queue will always be the node with the least heuristic.

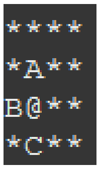
DFS: (Nodes added to the stack, left to right)

Root added to stack:



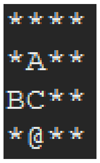
```
****
*A**
B@**
*C**
```

Popped from stack

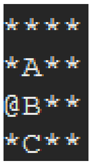


```
****
*A**
B@**
*C**
```

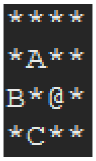
Nodes expanded from current node



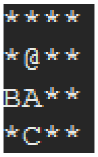
```
****
*A**
BC**
*@**
```



```
****
*A**
@B**
*C**
```

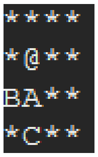


```
****
*A**
B*@*
*C**
```



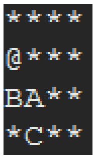
```
****
*@**
BA**
*C**
```

Popped from stack

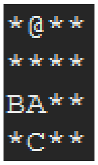


```
****
*@**
BA**
*C**
```

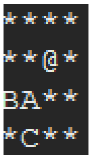
Nodes expanded from current node



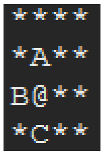
```
****
@***
BA**
*C**
```



```
*@**
****
BA**
*C**
```

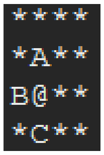


```
****
**@*
BA**
*C**
```



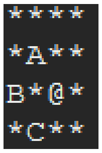
```
****
*A**
B@**
*C**
```

Popped from stack

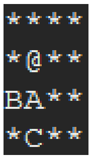


```
****
*A**
B@**
*C**
```

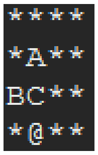
Nodes expanded from current node



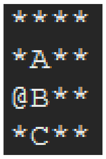
```
****
*A**
B*@*
*C**
```



```
****
*@**
BA**
*C**
```

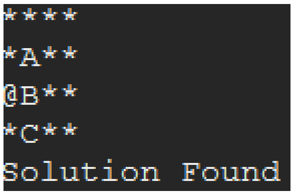


```
****
*A**
BC**
*@**
```



```
****
*A**
@B**
*C**
```

Popped from stack

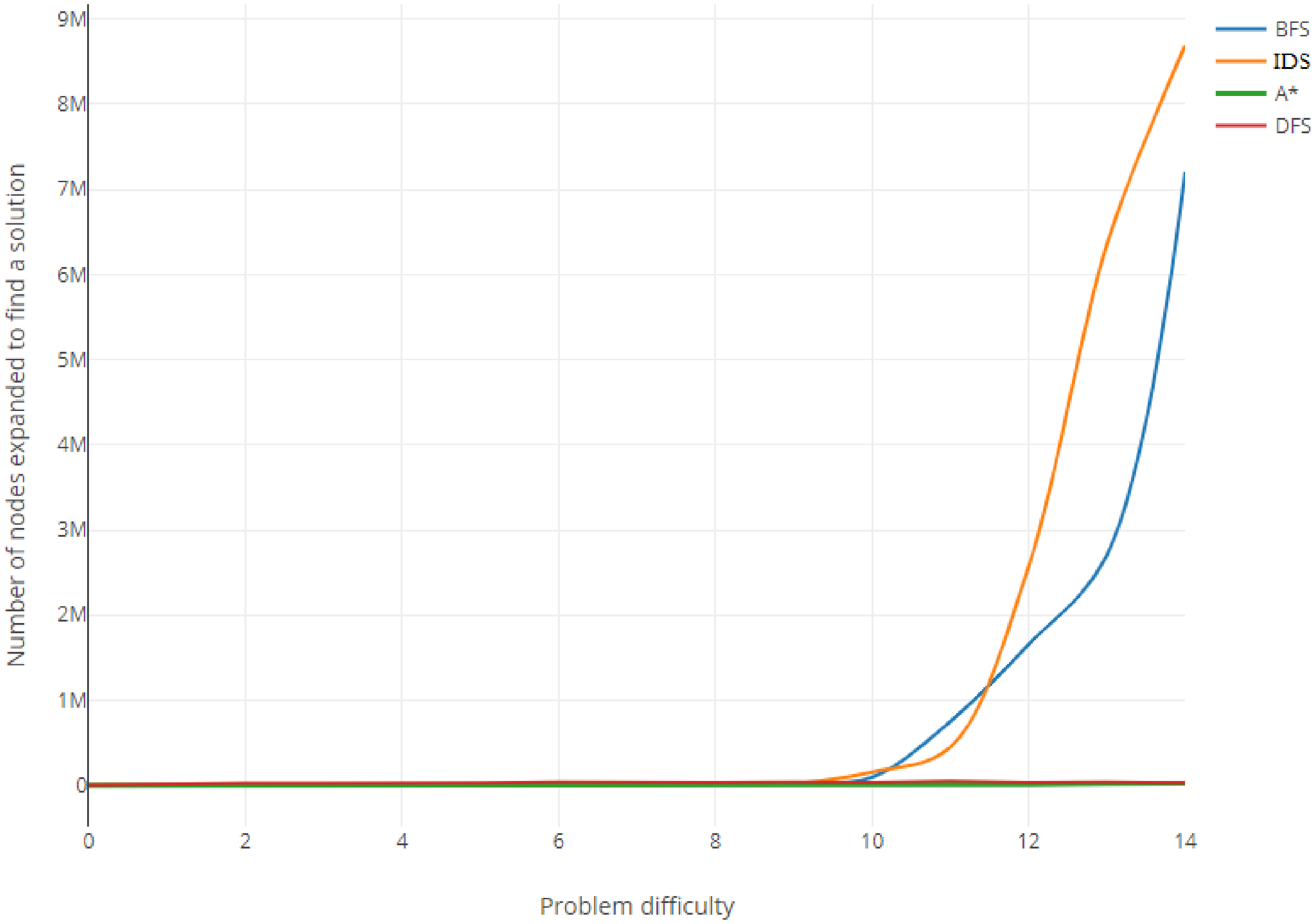


```
****
*A**
@B**
*C**
Solution Found
```

We can see that the next expanded node is always the last one expanded and respectively the last added to the stack. Another thing that should be noted here is that unlike the other examples where the order of expansion is always the same (right, left, up, down), here it is random each time. The purpose of that is to prevent DFS from entering a repeating loop.

Scalability Study

The way I decided to control problem difficulty is by controlling the depth of the solution. By using the graph search version of BFS I am finding the optimal path to the solution and then I am making each of the nodes from the path a starting state and recording the problem difficulty for it. By starting with the original configuration given in the problem, a solution is found at level 14 and that is why the range on the graph for problem difficulty is between 0 and 14 (i.e. it represents how many nodes are expanded when the start state is a certain number of steps away from the solution).



It can be observed that Iterative Deepening Search is the search method that expands the most nodes to find a solution, however, since it has the space complexity of DFS (*b*m, where b is the branching factor and m is the maximum depth; in our case the* *worst* *scenario would be 4*14 = 56 nodes at a time in memory*) it doesn't store too many nodes at a time and so it doesn't have a problem for finding a solution for larger problems although it can be slow.

When looking at the graph it can be seen that BFS expands a bit less nodes than IDS, however, it stores all of them in memory and thus it is more likely to run out of memory before it runs out of time compared to IDS.

Since I have randomised the order of expansions for DFS in order to avoid looping behaviour (the tree search version of DFS is incomplete), I ran each different starting state configuration 100 times and took the average number of nodes expanded.

On this graphing scale it might seem that DFS is nearly as good as A*, but if you actually look at the actual value it is evident that this isn't quite the case – DFS expands about 30 000 nodes (and that is thanks to randomising) whereas A* expands 26129 (both of these value are for cases when the start state is 14 steps away from the solution). A* uses the Manhattan distance heuristic to identify the next node.

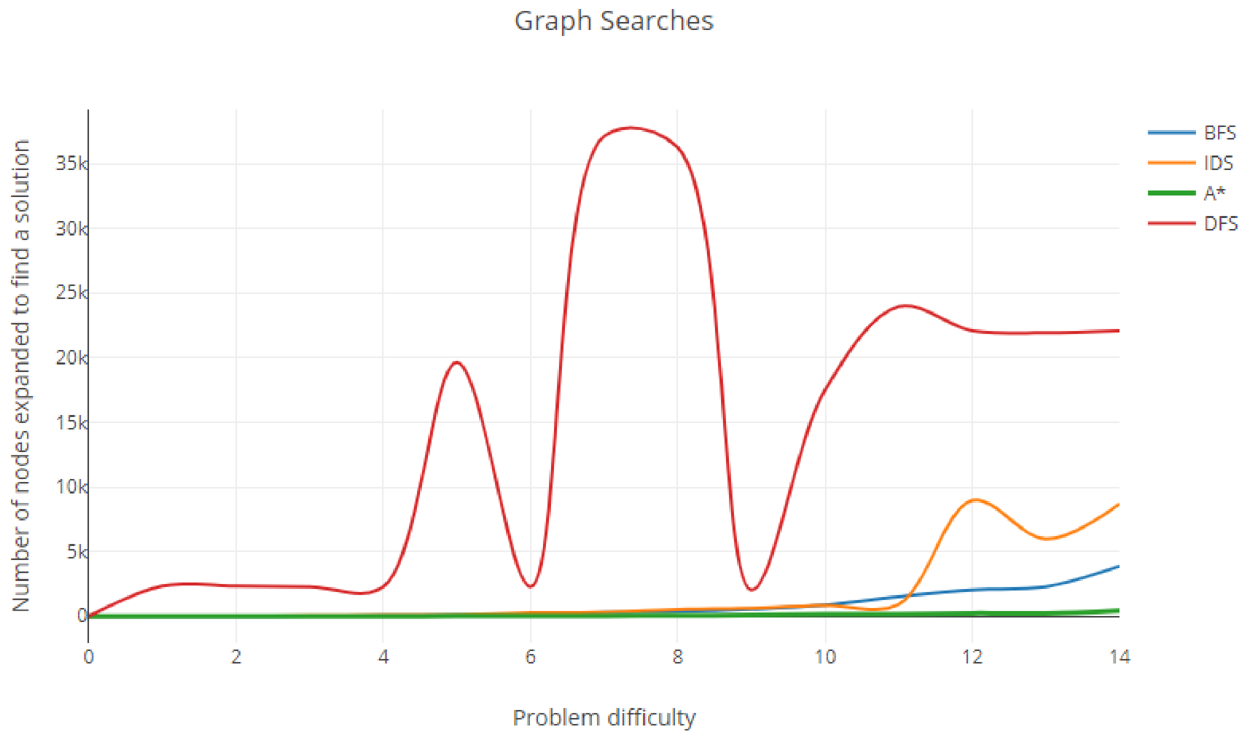
In the overall scheme of things A* is the optimal algorithm from the four to be performed on a tree search. A*, like IDS and BFS, is a complete algorithms (DFS is not). However, there might be a case when the search space for a large-scale problem is sufficiently large and then A* is likely to run out of space before it runs out of time since it stores all generated nodes in memory. This is the same problem that BFS has. In such case IDS might be a better choice, although it will expand way more nodes. DFS might not run out of memory as well for larger problems, but it is unlikely that it will find the optimal solution.

Extras and Limitations

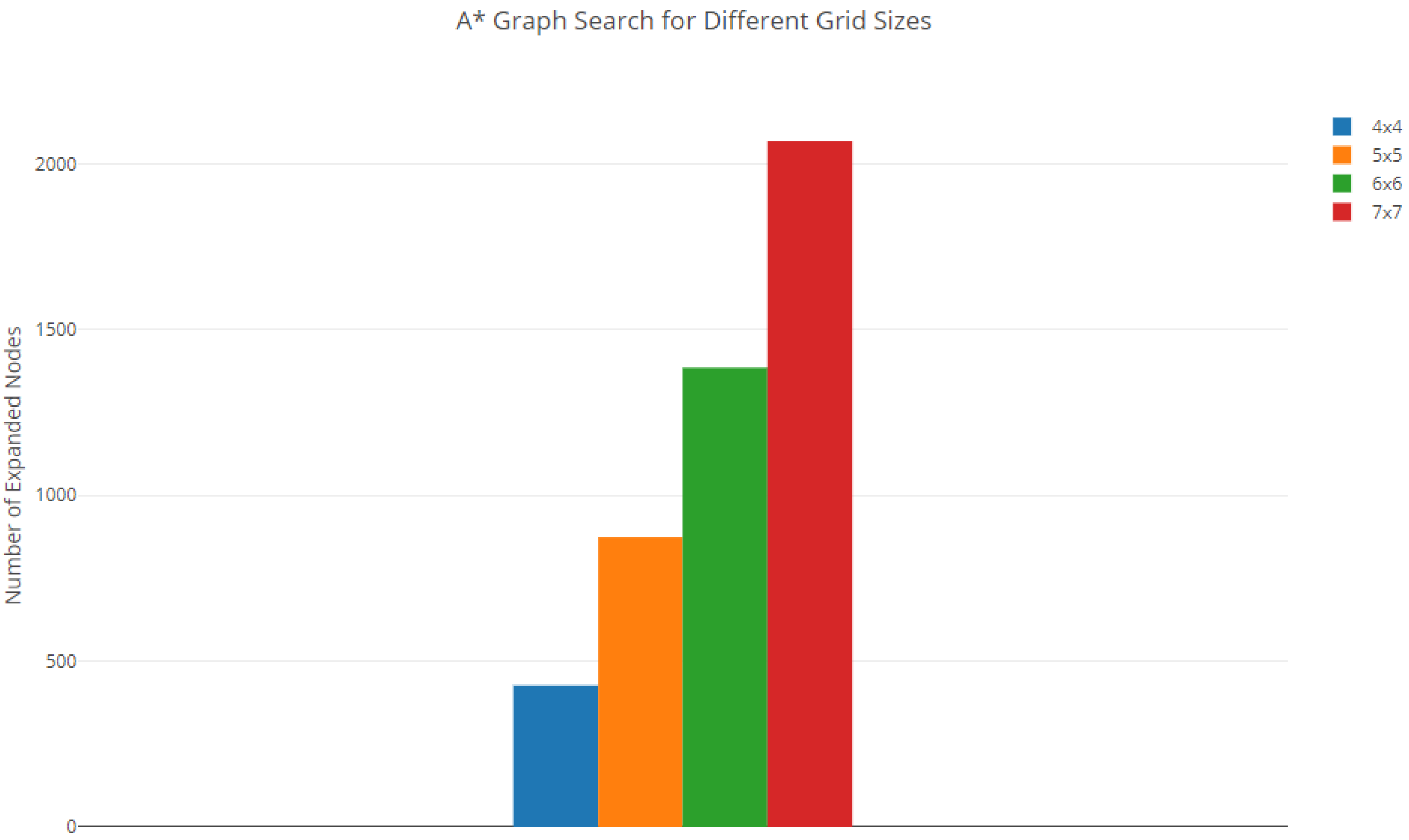
All the search methods can be improved significantly by being converted to their graph search equivalents, in other words they will avoid to expand configuration, which have already been checked.

As you can see on the following graph, A* still remains the best performing search algorithm, but the number of expanded nodes has decreased significantly – from 26129 to only 427 for the original starting state. BFS and DFS also see obvious improvements. For BFS the number of expanded nodes drops from around 7 million to 3870. For DFS the change is from almost 9 million to 8679 nodes.

Unlike the other searches, DFS doesn't really see a significant change. There are some lower values but they are only a few starting states that are closer to the final state. However, it is important to note that the graph version of DFS becomes complete (no need to randomise the next expanded node in this case), i.e. it founds a solution, although usually it is not the optimal one. The shape of the graph for DFS supports the fact that it rarely finds an optimal solution.



As another extension to my code I made the size of the grid to be resizable. To do that you simply need to change the values of the ‘row’ and ‘column’ variables in class *Node*. To illustrate this property the following graph represents how many nodes A* (in its graph search version) expands before it finds a solution. The start and end state are the same as the ones given in the definition of the original problem, but this time there are more configurations to be checked. The values given are respectively for grid size of 4x4, 5x5, 6x6, and 7x7.



An example of limitation to my work is for example the fact that I have hardcoded the start and final states. Also when I randomise DFS I ran each starting state configuration 100 times, if I had run it more times (say 1000) probably I would have gotten more accurate results for the scalability study.

Another thing that comes to my mind is that I could have used a one dimensional array to represent the grid instead of a two dimensional one. This way my code might have been more efficient in terms of memory usage. Moreover, the number of nodes generated to discover a solution is dependent on the order of which I am making the moves. In order to make the results as accurate as possible I am using the same sequence of steps for all searches (right, left, up, down). If I used another sequence I might have gotten a better or worse results (still the difference in the numbers wouldn't be that significant on the large scale).

Code

Node.java

```
import java.util.*;

public class Node {
    private int row = 4;
    private int col = 4;
    private char [][] positions = new char [row][col];
    private Node parent;

    private List<Node> children;
    private int agentRow;
    private int agentCol;
    private int level;

    public Node(char[][] positions) {
        this.positions = positions;
        this.children = new ArrayList<Node>();
    }

    public void allMoves(char[][] positions) {
        moveAgentRight(positions);
        moveAgentLeft(positions);
        moveAgentUp(positions);
        moveAgentDown(positions);
    }

    private void moveAgentRight(char[][] positions) {
        if(agentCol % col < col - 1) {
            char [][] newPositions = new char [row][col];
            makeCopy(positions, newPositions);

            char temp = newPositions[agentRow][agentCol + 1];
            newPositions[agentRow][agentCol + 1] =
newPositions[agentRow][agentCol];
            newPositions[agentRow][agentCol] = temp;

            Node child = new Node(newPositions);
            child.setAgentLocation(agentRow, agentCol + 1);
            child.parent = this;
            child.setLevel(child.parent.getLevel() + 1);
            children.add(child);
        }
    }

    private void moveAgentLeft(char[][] positions) {
        if(agentCol % col > 0) {
            char[][] newPositions = new char [row][col];
            makeCopy(positions, newPositions);

            char temp = newPositions[agentRow][agentCol - 1];
            newPositions[agentRow][agentCol - 1] =
newPositions[agentRow][agentCol];
            newPositions[agentRow][agentCol] = temp;
            Node child = new Node(newPositions);
            child.setAgentLocation(agentRow, agentCol - 1);
            child.parent = this;
            child.setLevel(child.parent.getLevel() + 1);
            children.add(child);
        }
    }

    private void moveAgentUp(char[][] positions) {
        if(agentRow % row > 0) {
            char[][] newPositions = new char [row][col];
            makeCopy(positions, newPositions);

            char temp = newPositions[agentRow - 1][agentCol];
            newPositions[agentRow - 1][agentCol] = newPositions
[agentRow][agentCol];
            newPositions[agentRow][agentCol] = temp;

            Node child = new Node(newPositions);
            child.setAgentLocation(agentRow - 1, agentCol);
            child.parent = this;
            child.setLevel(child.parent.getLevel() + 1);
            children.add(child);
        }
    }

    private void moveAgentDown(char[][] positions) {
        if(agentRow % row < row - 1) {
            char[][] newPositions = new char [row][col];
            makeCopy(positions, newPositions);

            char temp = newPositions[agentRow + 1][agentCol];
            newPositions[agentRow + 1][agentCol] =
newPositions[agentRow][agentCol];
            newPositions[agentRow][agentCol] = temp;

            Node child = new Node(newPositions);
            child.setAgentLocation(agentRow + 1, agentCol);
            child.parent = this;
            child.setLevel(child.parent.getLevel() + 1);
            children.add(child);
        }

    }

    public boolean gridExists(char [][] positions) {
        boolean isSame = true;
        for(int i=0; i<row; i++) {
            for(int j=0; j<col; j++) {
                if(this.positions[i][j] != positions[i][j]) {
                    isSame = false;
                }
            }
        }
        return isSame;
    }
    public boolean isFinalState() {

        boolean flag = false;
        if(this.positions[1][1] == 'A' && this.positions[2][1] == 'B' &&
this.positions[3][1] == 'C') {
            return true;
        }
        return flag;
    }

    private void makeCopy(char[][] a, char[][] b) {
        for (int i = 0; i < row; i++) {
            for(int j = 0; j < col; j++) {
                b[i][j] = a[i][j];
            }
        }
    }

    public void setAgentLocation(int r, int c) {
        this.agentRow = r;
        this.agentCol = c;
    }

    public void printGrid() {
        for(int i=0; i<positions.length; i++) {
            for(int j=0; j< positions.length; j++) {
                System.out.print(positions[i][j]);
            }
            System.out.println();
        }

    }

    public List<Node> getChildren(){
        return this.children;
    }

    public char[][] getPositions(){
        return this.positions;
    }

    public int getLevel() {
        return level;
    }

    public void setLevel(int level) {
        this.level = level;
    }

    public int manhattanDistance(char [][] positions) {
        int a = 0;
        int b = 0;
        int c = 0;

        for(int i=0; i<row; i++) {
            for(int j=0; j< col; j++) {
                if (positions[i][j] == 'A') {
                    a = Math.abs(1 - i) + Math.abs(1 - j);
                }
                if (positions[i][j] == 'B') {
                    b = Math.abs(2 - i) + Math.abs(1 - j);
                }
                if (positions[i][j] == 'C') {
                    c = Math.abs(3 - i) + Math.abs(1 - j);
                }
            }
        }
        return a + b + c;
    }

    public Node getParent() {
        return parent;
    }
}
```

Searches.java

```
import java.util.*;

public class Searches {

    public Searches() {

    }

    public void bFS (Node root) {
        List<Node> checked = new ArrayList<Node>();
        Queue<Node> q = new LinkedList<Node>();
        int cnt = 0;
        root.getChildren().clear();
        q.add(root);
        while(!q.isEmpty()) {
            ArrayList<Node> others = new ArrayList<Node>();
            Node node = q.poll();
            cnt++;
            checked.add(node);
            if (node.isFinalState()) {
                System.out.println(cnt);
                break;
            }

            node.allMoves(node.getPositions());

            for(Node n: node.getChildren()) {
                //this is the graph search version, in order to make it tree
search - remove the if condition
                if(!contained(checked, n)) {
                    q.add(n);
                }
            }
        }
    }
    private boolean contained(List<Node> checked, Node curr) {
        boolean flag = false;

        for(Node n: checked) {
            if(n.gridExists(curr.getPositions())) {
                flag = true;
            }
        }
        return flag;
    }

    public int dFS(Node root) {
        List<Node> checked = new ArrayList<Node>();
        Stack<Node> stack = new Stack<Node>();
        root.getChildren().clear();
        stack.push(root);
        int cnt = 0;
        while (!stack.isEmpty()) {
            Node node = stack.pop();
            cnt++;
            checked.add(node);
            if(node.isFinalState()) {
                break;
            }

            node.allMoves(node.getPositions());
            //Collections.shuffle(node.getChildren()); // randomizing the next
expanded node so that looping is prevented
            for(Node n: node.getChildren()) {
                if(!contained(checked,n)) {
        //this is the graph search version, in order to make it tree search -
remove the if condition and uncomment Collections.shuffle()
                    stack.push(n);
                }
            }
        }
        return cnt;
    }


    public int iDS(Node root) {
        List<Node> checked = new ArrayList<Node>();
        Stack<Node> stack = new Stack<Node>();
        stack.push(root);
        int cnt = 0;
        int limit = 0;
        while(!stack.isEmpty()) {
            Node node = stack.pop();
            checked.add(node);
            cnt++;

            if(node.isFinalState()) {
                break;
            }

            if(node.getLevel() < limit) {
                node.allMoves(node.getPositions());
                for(Node n: node.getChildren()) {
                    //this is the graph search version, in order to make it
tree search - remove the if condition
                    if(!contained(checked,n)) {
                        stack.push(n);
                    }
                }
            }

            if(stack.isEmpty()) {
                root.getChildren().clear();
                stack.push(root);
                checked.clear();
                limit++;
            }
        }
        return cnt;
    }

    public void aStar(Node root){
        List<Node> checked = new ArrayList<Node>();
        //new priority queue with custom comparator that compares nodes based
on their heuristic value
        PriorityQueue<Node> nodes = new PriorityQueue<>(11, (n1,n2) ->
heuristics(n1) - heuristics(n2));
        root.getChildren().clear();
        nodes.add(root);
        int cnt = 0;
        while(!nodes.isEmpty()) {
            Node node = nodes.poll();
            checked.add(node);
            cnt++;
            if(node.isFinalState()) {
                System.out.println(cnt);
                break;
            }

            node.allMoves(node.getPositions());
            for(Node n: node.getChildren()) {
                //this is the graph search version, in order to make it tree
search - remove the if condition
                if(!contained(checked,n)) {
                    nodes.add(n);
                }
            }
        }
    }

    private int heuristics(Node n) {
        return n.getLevel() + n.manhattanDistance(n.getPositions());
    }

}
```

Main.java

```
/**
 * If you want to run any of the searches just uncomment the corresponding one
 * If you want to see the path to the nearest solution uncomment the list,
which stores it
 */
import java.util.*;

public class Main {


    public static void main(String[] args) {
        char[][] startState = {
                {'*','*','*','*'},
                {'*','*','*','*'},
                {'*','*','*','*'},
                {'A','B','C','@'}
        };
        Node root = new Node(startState);
        root.setAgentLocation(3,3);
        root.setLevel(0);

        Searches search = new Searches();
        //List<Node> path = findPath(root);

        //search.bFS(root);
        //search.dFS(root);
        //search.iDS(root);
        //search.aStar(root);



    }

    private static List<Node> findPath(Node root){
        List<Node> path = new ArrayList<>();
        List<Node> checked = new ArrayList<Node>();
        Queue<Node> q = new LinkedList<Node>();
        int cnt = 0;
        q.add(root);
        while(!q.isEmpty()) {
            Node node = q.poll();

            cnt++;
            checked.add(node);
            if (node.isFinalState()) {
                int lvl = node.getLevel();
                for (int i =0; i < lvl + 1; i++) {
                    path.add(node);
                    node = node.getParent();
                }
                System.out.println("Solution Found!");
                break;
            }

            node.allMoves(node.getPositions());
```

```java
            for(Node n: node.getChildren()) {
                if(!contained(checked, n)) {
                    q.add(n);
                }
            }
        }
        return path;
    }

    private static boolean contained(List<Node> checked, Node curr) {
        boolean flag = false;

        for(Node n: checked) {
            if(n.gridExists(curr.getPositions())) {
                flag = true;
            }
        }
        return flag;
    }
}
```