University of Southampton

COMP2208

Intelligent Systems

Search Methods Coursework Assignment

Alked Ejupi
ae7g18@soton.ac.uk

November 2019

Contents

1 Approach

1.1 Formulation of the blocks world puzzle

Before the actual implementation of the search algorithms to solve the blocks world puzzle, I have formulated the **problem** in order to achieve generalisation in terms of further scalability in the future. The first thing I have defined is the ***state*** of a board. This contains a description of the board in a particular time where it specifies the positions of each tile A, B, C and the Agent in a grid of size NxN ($4x4$ in the case of the problem given). Once defined the structure of a single state, it was possible to formally describe the **initial state** describing the given initial configuration of the board. Another important aspect was specifying what were the ***Actions*** of the agent that could be done on the board, this refers to the agent moves ***up***, ***down***, ***left*** and ***right***. In order to apply the move of an agent in a board, I have also defined the **transition model** where given a particular state and an action it returns a new state with the movement of the agent applied, obtaining a new configuration of the board.

1.2 Environment setup

I chose the *Java* language to implement the blocks world and the search algorithms, as it is the language which I am most proficient with. I organised my Java project in way that I could easily add features without having the trouble to refactor the code or apply enormous changes to the existing code.The `Utils.java` and `Debug.java` class contain methods that are used across the entire project such as printing the state of the board in ASCII, common operations such as converting an array 2D to 1D, creating a 2D array of cells or printing debug statements.

1.3 Defining the problem

All the code related to the problem comes under the package named `Problem`. I decided to keep it as abstract as possible, for this reason I defined an interface called `Problem.java` which includes all the information that the blocks world puzzle should have in order to be solved.

1.4 Implementing the blocks world

All code related to the blocks world can be found under the package `BlocksWorld`. This includes the class `Cell` which defines a single cell of the board, it has also an inner enum class called `CellType` representing the types of cells, tile A, B and C, *Agent* and *empty* cell. The class `Board` describes the internal structure of a board, that is, a two dimensional array of cells (type `Cell`). The board contains also the position (type `Point`) of each tile (A, B and C) and the Agent.

1.5 Implementing the search methods

I have created an abstract class called `TreeSearch.java`. This contains common operations and properties that search trees have, such as creating a root node, generating successors nodes, return a solution given a node, tracking the number of nodes generated and an abstact search tree method which refers to the strategy we want to use. The abstract search tree method (`List<Node> search(Problem problem)`) is implemented in `BFS.java`, `DFS.java`, `IDS.java` and `AStar.java` according to the strategy they have.

The class `Node.java` contains the attributes required to keep track of the tree we are constructing. These are the current `State`, the parent `Node`, the `Action` that was taken by the parent and the `pathCost` computed by summing all step cost from the initial state to the node and its depth level. This class also implements `Heuristic.java` interface required to implement A*.

1.5.1 BFS, DFS and A*Star

These three search methods are exactly the same in terms of procedure, the only that changes is the data structure for storing all the nodes that need to be expanded, this is called *fringe* in the code. BFS uses a ***FIFO*** *queue* `Queue<Node> in the code` using for adding and removing, DFS uses a ***LIFO*** *stack* (`Stack<Node>`) and A* uses a priority queue (`PriorityQueue<Node>`) taking as argument a comparator (`Comparator.comparingInt(node -> node.estimatedCost)`) that compares nodes using their estimated cost. Another slight difference is that in DFS, before adding all successors to the fringe they are first shuffled in order to avoid loops. This is done by using the *Java* method `Collections.shuffle(aList)`.

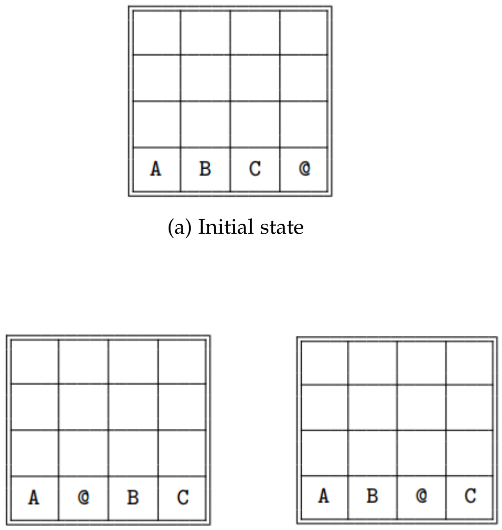
1.5.2 IDS

In iterative deepening I decided to use a recursive approach. It works slightly different from the others, it needs a finite limit (here I chose `LIMIT = Integer.MAX_VALUE`) which defines the maximum depth at which the solution can be found. The depth limit starts at $d = 0$ and at each iteration calls depth limited search (DFS), then if the returned node is not `null` increments $d + 1$ otherwise return node as the solution. DFS will call its self with depth limit equal to d calling recursively its self again for each successor generated up to the depth limit specified. The recursive call will terminate when reached the limit or a solution has been found. IDS will terminate if DFS finds a solution or no solution has been found for all depth limits available.

2 Evidence

The evidence related to the correctness of each implementation can be found in evidence appendix A. Each section represent a detailed debug output of each search methods. I decided to set an easy version of the problem for showing all the steps that each algorithm does. For A*, DFS and IDS I choose Goal A (depth 2), while for BFS I chose the problem to be Goal B (depth 1). Both problems have the same initial state.

In the evidence appendix A the configuration of a state is represented as a string. For example, the configuration of initial state in Fig. 2 (a) is `------------ABC@`



(a) Initial state

(b) Goal A

(c) Goal B

Figure 1: Problem A & B

2.1 BFS Evidence

The detailed debug output (A.1) shows how **BFS** expands the nodes in tree. This can be noticed by how **BFS** picks the first node to expand from the **Queue** (*FIFO*). Once it is picked and removed from the *fringe*, it checks whether it is the solution and if it is not, it will expand the node and add all the *successors*. The process will be repeated until it finds the solution. Appendix A 1 shows all steps.

2.2 DFS Evidence

Appendix A.2 shows how **DFS** expands the nodes in tree. Since **DFS** uses a Stack (*LIFO*) for the fringe, we can clearly see that every time that removes a node from the *fringe*, it always picks the last inserted. The successors generated are always shuffled before added to the *fringe*. The output clearly shows in some steps that the order of the *successors* are not the same as the order they are inserted. The fact that the number of the steps to reach the goal is 4 implies that the moves are purely random, giving us a non optimum solution. This shows that **DFS** is implemented correctly.

2.3 AStar Evidence

A* uses a **Priority Queue** for the fringe. Every time that a nodes is created, an *estimated cost* is computed. The priority queue compares their *estimated cost* and it chooses the node with the lower cost. If two or more nodes have the same cost, it picks the one added first. The debug (A.3) shows step by step how the algorithm expands the node with lower cost present in *fringe*.

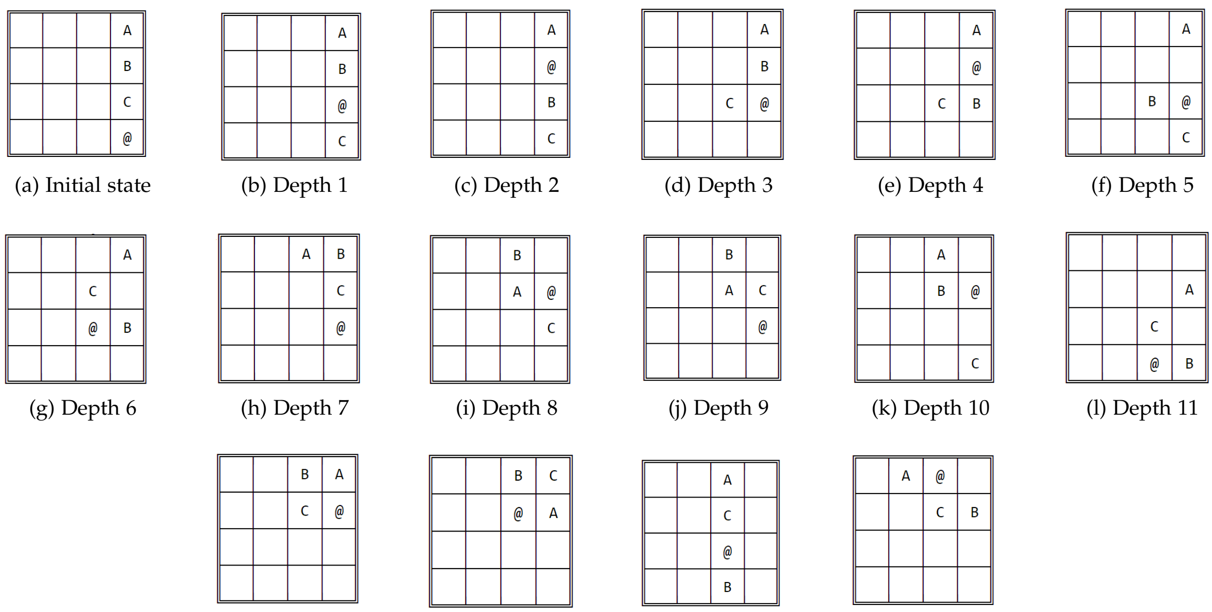
2.4 IDS Evidence

To show that the **IDS** implementation is correct is a bit trickier. Since it is uses recursion, there is no *fringe* to store the nodes to expanded. In the debug output there is information about when **DLS** (recursive) is called and when when the depth limit changes. **DLS** is called recursively on every *successor* generated and it stops when it reaches the *limit*, starting a new iteration in **IDS** changing the depth limit to $d + 1$. The debug output shows exactly how this is done. (Appendix A.4)

3 Scalability Study

For the scalability study I decided to measure the time complexity by using 15 different problems using the depth to measure the difficulty, the initial state remains the same. (see Fig. 3). The way I picked all these problems is by using **BFS** to explore all the nodes generated up to depth 15 and adding all those nodes generated in a *set*, I then printed out all the nodes with unique configuration along with its depth.

This made sure that the problem was unique and could not be solved by a different depth level. For instance, a specific configuration can be found in both depth level 3 and level 6 which means that the real difficulty of the problem is actually 3 and not 6.

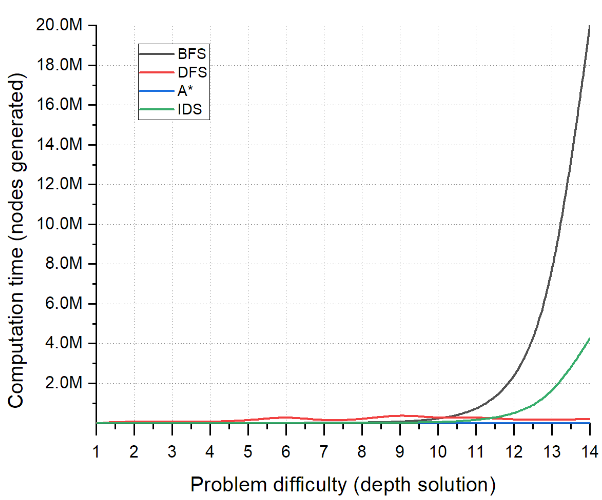


(a) Initial state (b) Depth 1 (c) Depth 2 (d) Depth 3 (e) Depth 4 (f) Depth 5

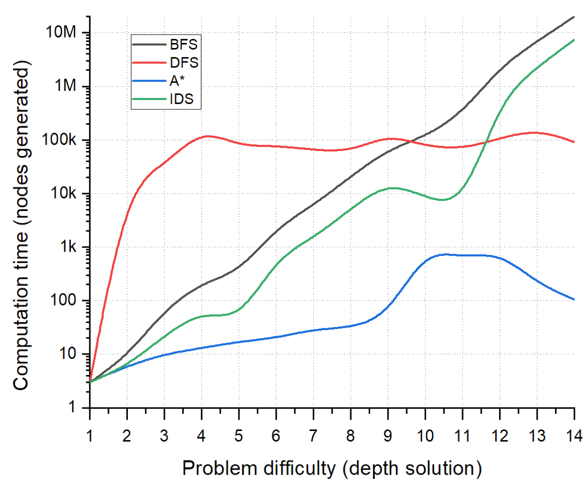
(g) Depth 6 (h) Depth 7 (i) Depth 8 (j) Depth 9 (k) Depth 10 (l) Depth 11

(m) Depth 12 (n) Depth 13 (o) Depth 14 (p) Depth 15

Figure 2: 15 problems



(a) linear graph



(b) semi-log graph

Figure 3: Time complexity (nodes generated vs problem difficulty)

The graphs in Figure 3 illustrates the time complexity of each algorithm, the horizontal axis represents the problem difficulty by the depth level, while the vertical axis shows the number of nodes generated in the tree.

3.1 IDS & BFS time complexity

If we look closely at **IDS** and **BFS**, we can clearly see that they have the same exponential growth, this matches the expected time complexity $O(b^d)$ which they both have in common. However, **IDS** generates less nodes than **BSF**, the reason is because **BFS** searches for the solution from *right* to *left* and then goes to the next level, while IDS from *top* to *bottom* and then *right* to *left*. **IDS** is better for finding solutions placed at bottom of the tree, especially when it is placed on the bottom left. Those solutions require less generated nodes for **IDS**, while **BFS** has to generate all the nodes of each level before reaching the desired depth where the solution can be found. This explains why **IDS** generated less nodes in some points, rather than always growing exponentially. Overall, we can also say that they both scan the entire tree finding a solution if exists meaning that they are both *complete*.

3.2 DFS time complexity

DFS search strategy can be easily noticed from diagram, as the number of nodes generated at each depth level is not directly related to the difficulty level. The number of nodes for easy problems is huge compared to the other strategies. This tells us that it does not find an optimum solution and it wastes a lot of time expanding the deepest nodes in a random way until it finds a solution.

3.3 A* time complexity

A* is the best amongst the others. The main reason is because is the only informed search strategy. The rate of growth is relatively slower than the the others as it chooses least amount of nodes to be expanded by using the cost of the evaluation function. The graph shows also that my heuristic is not admissible, this can be seen at the problem difficulty from 10 to 12 where the nodes generated grow rapidly before decreasing again at difficulty 15.

3.4 Conclusion

To sum up, **A*** is the best algorithm in terms of time complexity, however since it is not admissible, it does not always find an optimum solution. For this reason **IDS** and **BFS** are better for finding optimum solution, although **IDS** is preferred for its better time complexity as **BFS** will sometimes need to generate a huge amount of nodes before reaching the solution. Despite the fact that **DFS** sometimes provides an acceptable time complexity, the solution found is not optimum at all.

4 Limitations & Extra

4.1 Limitations

4.1.1 Memory optimisation

My code uses a lot of memory, especially when I run BFS I need to change the heap space to 15GB by changing the JVM setting. This is due to the fact that I am generating a full grid of `Cells` incuding empty cells every time a `Node` is generated. One solution might be just storing the value of the tiles and agent without using a 2D array which wastes a lot of memory.

4.1.2 Code Duplication

I could have reduced the amount of code for the implementation of these search methods to just one class. The class would represent a general search for trees which takes as argument the type of the fringe, that is, FIFO queue, LIFO queue and Priority Queue. For IDS the improved class would be slightly modified by using an iterative approach and not recursive.

4.2 Extra

4.2.1 Heuristic for A*

The first implementation of my heuristic was admissible since it was using the Manhattan distance and the path cost. My improved version is no longer admissible since in some cases as shown in the scalability study overestimates the costs of some nodes. The heuristic uses the Manhattan distance, the path cost, the cost of misplaced tiles and the cost of the agent to get to the misplaced tiles. In appendix A.5 and A.6 can be found the solution of the non admissible and admissible heuristic respectively. For the problem given in the specification, my heuristic just generates 96 nodes, while the admissible solution generates 56007 nodes.

Appendices

A Evidence Appendix

A.1 BFS

BFSOutput.txt

```
INITIAL STATE
```

A	B	C	@

```
GOAL STATE
```

A	B	@	C

```
I'm solving the puzzle with BFS...
Adding root ------------ABC@to the fringe.
Removing node ------------ABC@ from the fringe

Check if ------------ABC@ is the goal...
It is not the goal,  then expand node ------------ABC@

Start expanding node ------------ABC@

 -----------@ABC-
Action taken: UP
```

			@
A	B	C	

```
 ------------AB@C
Action taken: LEFT
```

A	B	@	C

```
End expansion of ------------ABC@
No. successors generated: 2

Adding 2 successors to the fringe.
     FRINGE
```

-----------@ABC-	1
------------AB@C	2

```
Removing node -----------@ABC- from the fringe
```

```
Check if -----------@ABC- is the goal...
It is not the goal,  then expand node -----------@ABC-

Start expanding node -----------@ABC-

 -------@----ABC-
Action taken: UP
```

			@
A	B	C	

```
 ------------ABC@
Action taken: DOWN
```

A	B	C	@

```
 ----------@-ABC-
Action taken: LEFT
```

		@	
A	B	C	

```
End expansion of -----------@ABC-
No. successors generated: 3

Adding 3 successors to the fringe.
     FRINGE
```

FRINGE	
------------AB@C	1
-------@----ABC-	2
------------ABC@	3
----------@-ABC-	4

```
Removing node ------------AB@C from the fringe

Check if ------------AB@C is the goal...
Node ------------AB@C is the goal!

Solution:

Step 1: LEFT
Configuration: ------------AB@C
```

A	B	@	C

```
Time elapsed: 78ms
Number nodes generated: 6
Depth solution : 1
```

```
Moves: LEFT
```

A.2 DFS

DFSOutput.txt

```
INITIAL STATE
```

A	B	C	@

```
GOAL STATE
```

A	@	B	C

```
I'm solving the puzzle with DFS...

Adding root ------------ABC@to the fringe.

     FRINGE
```

------------ABC@	1

```
Removing node ------------ABC@ from the fringe

Check if ------------ABC@ is the goal...
It is not the goal,  then expand node ------------ABC@

Start expanding node ------------ABC@

 -----------@ABC-
Action taken: UP
```

			@
A	B	C	

```
 ------------AB@C
Action taken: LEFT
```

A	B	@	C

```
End expansion of ------------ABC@
No. successors generated: 2

Shuffling the order of the successors...
```

```
Adding 2 successors to the fringe.
     FRINGE
-----------@ABC-  1
------------AB@C  2

Removing node ------------AB@C from the fringe

Check if ------------AB@C is the goal...
It is not the goal,  then expand node ------------AB@C

Start expanding node ------------AB@C

 ----------@-AB-C
Action taken: UP
```

		@	
A	B		C

```
 ------------A@BC
Action taken: LEFT
```

A	@	B	C

```
 ------------ABC@
Action taken: RIGHT
```

A	B	C	@

```
End expansion of ------------AB@C
No. successors generated: 3

Shuffling the order of the successors...
Adding 3 successors to the fringe.
     FRINGE
-----------@ABC-  1
----------@-AB-C  2
------------A@BC  3
------------ABC@  4

Removing node ------------ABC@ from the fringe

Check if ------------ABC@ is the goal...
It is not the goal,  then expand node ------------ABC@

Start expanding node ------------ABC@

 -----------@ABC-
Action taken: UP
```


			@
A	B	C	

```
 ------------AB@C
Action taken: LEFT
```

A	B	@	C

```
End expansion of ------------ABC@
No. successors generated: 2

Shuffling the order of the successors...
Adding 2 successors to the fringe.
     FRINGE
```

-----------@ABC-	1
----------@-AB-C	2
------------A@BC	3
-----------@ABC-	4
------------AB@C	5

```
Removing node ------------AB@C from the fringe

Check if ------------AB@C is the goal...
It is not the goal,  then expand node ------------AB@C

Start expanding node ------------AB@C

 ----------@-AB-C
Action taken: UP
```

		@	
A	B		C

```
 ------------A@BC
Action taken: LEFT
```

A	@	B	C

```
 ------------ABC@
Action taken: RIGHT
```

A	B	C	@

```
End expansion of ------------AB@C
No. successors generated: 3

Shuffling the order of the successors...
```

```
Adding 3 successors to the fringe.
     FRINGE
-----------@ABC-  1
----------@-AB-C  2
------------A@BC  3
-----------@ABC-  4
------------ABC@  5
----------@-AB-C  6
------------A@BC  7
Removing node ------------A@BC from the fringe

Check if ------------A@BC is the goal...
Node ------------A@BC is the goal!

Solution:
11
Step 1: LEFT
Configuration: ------------AB@C
```

A	B	@	C

```
Step 2: RIGHT
Configuration: ------------ABC@
```

A	B	C	@

```
Step 3: LEFT
Configuration: ------------AB@C
```

A	B	@	C

```
Step 4: LEFT
Configuration: ------------A@BC
```

A	@	B	C

```
Time elapsed: 39ms
Number nodes generated: 11
Depth solution : 4
Moves: LEFT RIGHT LEFT LEFT

Process finished with exit code 0
```

A.3 IDS

IDSOutput.txt

INITIAL STATE

A	B	C	@

GOAL STATE

A	@	B	C

```
I'm solving the puzzle with IDS...

Creating root with initial state: ------------ABC@

Performing IDS at depth limit 0.

Performing recursive DLS calls d=0 on root node ------------ABC@
Check if ------------ABC@ is the goal...
It is not the goal,  then expand node ------------ABC@

End of recursive DLS calls with d=0

Solution not found at depth limit 0.
End performing IDS at depth limit 0.

Performing IDS at depth limit 1.

Performing recursive DLS calls d=1 on root node ------------ABC@
Check if ------------ABC@ is the goal...
It is not the goal,  then expand node ------------ABC@

Start expanding node ------------ABC@

 -----------@ABC-
Action taken: UP
```

			@
A	B	C	

```
 ------------AB@C
Action taken: LEFT
```

A	B	@	C

```
End expansion of ------------ABC@
No. successors generated: 2
```

```
Calling DFS on 2 successors

Performing recursive DLS on -----------@ABC-
Check if -----------@ABC- is the goal...
It is not the goal,  then expand node -----------@ABC-

Performing recursive DLS on ------------AB@C
Check if ------------AB@C is the goal...
It is not the goal,  then expand node ------------AB@C

End of recursive DLS calls with d=1

Solution not found at depth limit 1.
End performing IDS at depth limit 1.

Performing IDS at depth limit 2.

Performing recursive DLS calls d=2 on root node ------------ABC@
Check if ------------ABC@ is the goal...
It is not the goal,  then expand node ------------ABC@

Start expanding node ------------ABC@

 -----------@ABC-
Action taken: UP
```

			@
A	B	C	

```
 ------------AB@C
Action taken: LEFT
```

A	B	@	C

```
End expansion of ------------ABC@
No. successors generated: 2

Calling DFS on 2 successors

Performing recursive DLS on -----------@ABC-
Check if -----------@ABC- is the goal...
It is not the goal,  then expand node -----------@ABC-

Start expanding node -----------@ABC-

 -------@----ABC-
Action taken: UP
```

			@
A	B	C	

```
 ------------ABC@
Action taken: DOWN
```


A	B	C	@

```
 ----------@-ABC-
Action taken: LEFT
```

		@	
A	B	C	

```
End expansion of -----------@ABC-
No. successors generated: 3

Calling DFS on 3 successors

Performing recursive DLS on -------@----ABC-
Check if -------@----ABC- is the goal...
It is not the goal,  then expand node -------@----ABC-

Performing recursive DLS on ------------ABC@
Check if ------------ABC@ is the goal...
It is not the goal,  then expand node ------------ABC@

Performing recursive DLS on ----------@-ABC-
Check if ----------@-ABC- is the goal...
It is not the goal,  then expand node ----------@-ABC-

Performing recursive DLS on ------------AB@C
Check if ------------AB@C is the goal...
It is not the goal,  then expand node ------------AB@C

Start expanding node ------------AB@C

 ----------@-AB-C
Action taken: UP
```

		@	
A	B		C

```
 ------------A@BC
Action taken: LEFT
```

A	@	B	C

```
 ------------ABC@
Action taken: RIGHT
```

A	B	C	@

```
End expansion of ------------AB@C
No. successors generated: 3

Calling DFS on 3 successors

Performing recursive DLS on ----------@-AB-C
Check if ----------@-AB-C is the goal...
It is not the goal,  then expand node ----------@-AB-C

Performing recursive DLS on ------------A@BC
Check if ------------A@BC is the goal...
Ending recursive DLS at ------------A@BC
Solution found, return value to IDS.

End of recursive DLS calls with d=2

Solution found at d=2
Step 1: LEFT
Configuration: ------------AB@C
```

A	B	@	C

```
Step 2: LEFT
Configuration: ------------A@BC
```

A	@	B	C

```
Time elapsed: 94ms
Number nodes generated: 9
Depth solution : 2
Moves: LEFT LEFT

Process finished with exit code 0
```

A.4 AStar

AStarOutput.txt

```
INITIAL STATE
```

A	B	C	@

```
GOAL STATE
```


A	@	B	C

```
I'm solving the puzzle with AStar...

Adding root ------------ABC@to the fringe.

     FRINGE
```

------------ABC@	cost=9

```
Removing node ------------ABC@ from the fringe

     FRINGE
```

empty

```
Check if ------------ABC@ is the goal...
It is not the goal,  then expand node ------------ABC@

Start expanding node ------------ABC@

 -----------@ABC-
estimated cost : 12
Action taken: UP
```

			@
A	B	C	

```
 ------------AB@C
estimated cost : 7
Action taken: LEFT
```

A	B	@	C

```
End expansion of ------------ABC@
No. successors generated: 2

Adding 2 successors to the fringe.
     FRINGE
```

------------AB@C	cost=7
-----------@ABC-	cost=12

```
Removing node ------------AB@C from the fringe

     FRINGE
```

-----------@ABC-	cost=12

```
Check if ------------AB@C is the goal...
It is not the goal,  then expand node ------------AB@C

Start expanding node ------------AB@C

 ----------@-AB-C
estimated cost : 9
Action taken: UP
```


		@	
A	B		C

```
 ------------A@BC
estimated cost : 2
Action taken: LEFT
```

A	@	B	C

```
 ------------ABC@
estimated cost : 11
Action taken: RIGHT
```

A	B	C	@

```
End expansion of ------------AB@C
No. successors generated: 3

Adding 3 successors to the fringe.
     FRINGE
```

------------A@BC	cost=2
------------ABC@	cost=11
----------@-AB-C	cost=9
-----------@ABC-	cost=12

```
Removing node ------------A@BC from the fringe

     FRINGE
```

----------@-AB-C	cost=9
------------ABC@	cost=11
-----------@ABC-	cost=12

```
Check if ------------A@BC is the goal...

Node ------------A@BC is the goal!

Solution:

Step 1: LEFT
Configuration: ------------AB@C
```

A	B	@	C

```
Step 2: LEFT
Configuration: ------------A@BC
```


A	@	B	C

```
Time elapsed: 47ms
Number nodes generated: 6
Depth solution : 2
Moves: LEFT LEFT

Process finished with exit code 0
```

A.5 Non admissible heuristic solution

nonAdmissibleHeuristic.txt

```
INITIAL STATE
```

A	B	C	@

```
GOAL STATE
```

	A		
@	B		
	C		

```
I'm solving the puzzle with AStar...
Step 1: UP
Configuration: -----------@ABC-
```

			@
A	B	C	

```
Step 2: LEFT
Configuration: ----------@-ABC-
```

		@	
A	B	C	

```
Step 3: LEFT
Configuration: ---------@--ABC-
```


	@		
A	B	C	

```
Step 4: DOWN
Configuration: ---------B--A@C-
```

	B		
A	@	C	

```
Step 5: LEFT
Configuration: ---------B--@AC-
```

	B		
@	A	C	

```
Step 6: UP
Configuration: --------@B---AC-
```

@	B		
	A	C	

```
Step 7: RIGHT
Configuration: --------B@---AC-
```

B	@		
	A	C	

```
Step 8: DOWN
Configuration: --------BA---@C-
```

B	A		
	@	C	

```
Step 9: RIGHT
Configuration: --------BA---C@-
```

B	A		
	C	@	

```
Step 10: UP
Configuration: --------BA@--C--
```

B	A	@	
	C		

```
Step 11: UP
Configuration: ------@-BA---C--
```

		@	
B	A		
	C		

```
Step 12: LEFT
Configuration: -----@--BA---C--
```

	@		
B	A		
	C		

```
Step 13: DOWN
Configuration: -----A--B@---C--
```

	A		
B	@		
	C		

```
Step 14: LEFT
Configuration: -----A--@B---C--
```

	A		
@	B		
	C		

```
Time elapsed: 0ms
Number nodes generated: 96
Depth solution : 14
Moves: UP LEFT LEFT DOWN LEFT UP RIGHT DOWN RIGHT UP UP LEFT DOWN LEFT
```

A.6 Admissible heuristic solution

admissibleHeuristic.txt

```
INITIAL STATE
```


A	B	C	@

```
GOAL STATE
```

	A		
@	B		
	C		

```
I'm solving the puzzle with AStar...
Step 1: UP
Configuration: -----------@ABC-
```

			@
A	B	C	

```
Step 2: LEFT
Configuration: ----------@-ABC-
```

		@	
A	B	C	

```
Step 3: LEFT
Configuration: ---------@--ABC-
```

	@		
A	B	C	

```
Step 4: DOWN
Configuration: ---------B--A@C-
```

	B		
A	@	C	

```
Step 5: LEFT
Configuration: ---------B--@AC-
```

	B		

@	A	C	

Step 6: UP
Configuration: --------@B---AC-

@	B		
	A	C	

Step 7: RIGHT
Configuration: --------B@---AC-

B	@		
	A	C	

Step 8: DOWN
Configuration: --------BA---@C-

B	A		
	@	C	

Step 9: RIGHT
Configuration: --------BA---C@-

B	A		
	C	@	

Step 10: UP
Configuration: --------BA@--C--

B	A	@	
	C		

Step 11: UP
Configuration: ------@-BA---C--

		@	
B	A		
	C		

Step 12: LEFT
Configuration: -----@--BA---C--

	@		
B	A		
	C		

Step 13: DOWN
Configuration: -----A--B@---C--

	A		
B	@		
	C		

Step 14: LEFT
Configuration: -----A--@B---C--

	A		
@	B		
	C		

Time elapsed: 328ms
Number nodes generated: 56007
Depth solution : 14
Moves: UP LEFT LEFT DOWN LEFT UP RIGHT DOWN RIGHT UP UP LEFT DOWN LEFT

B Appendix Code

The way I structured the code can be seen in the below diagram;

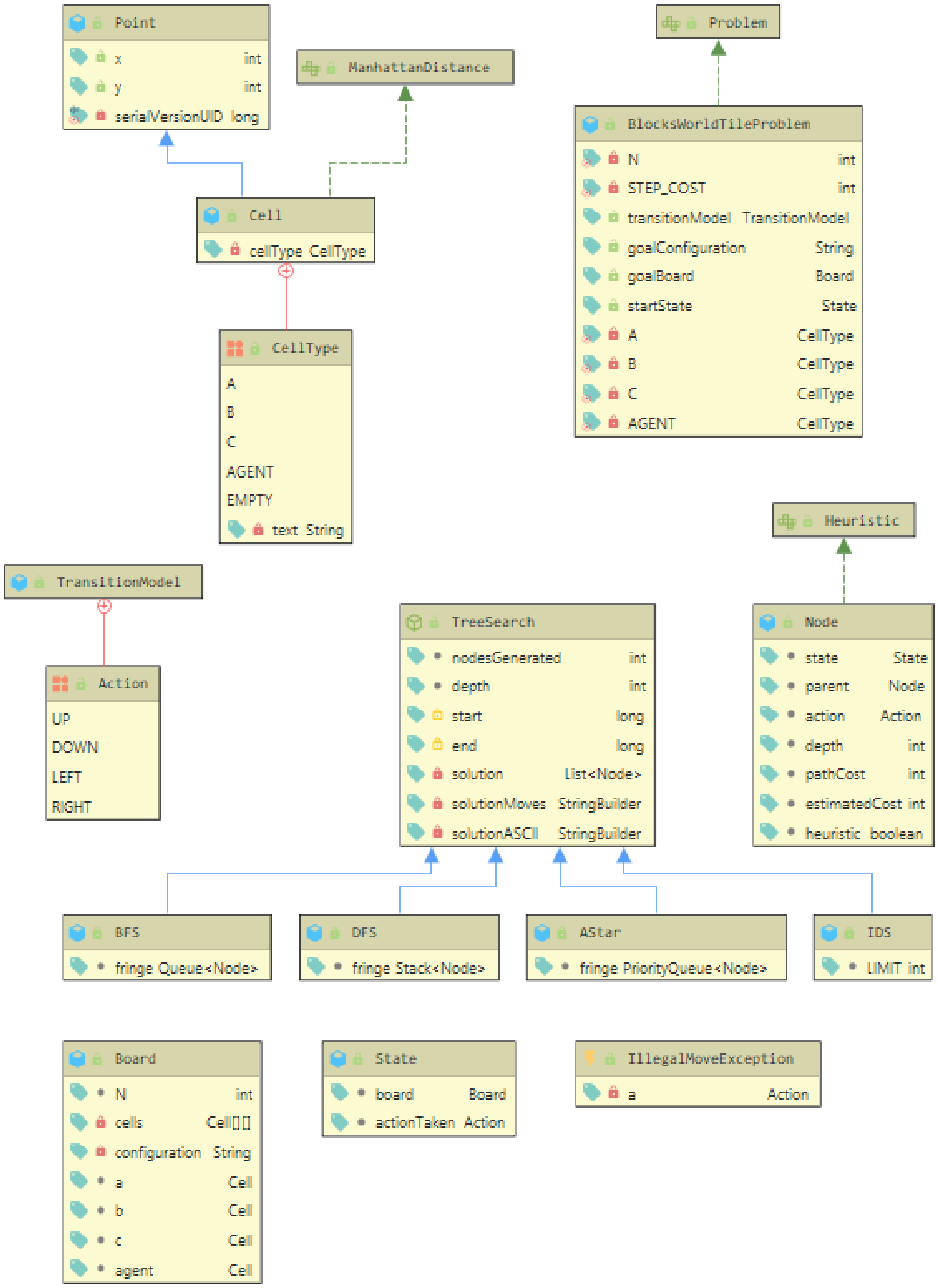


Figure 4: Class diagram

B.1 `TreeSearchAlgorithms` package

```
package TreeSearchAlgorithm;

import Exceptions.IllegalMoveException;
import Problem.Problem;
import Problem.State;
import Utils.Debug;
import java.util.*;
```

```java
import Problem.TransitionModel.Action;
import Utils.Utils;

/**
 * Abstract class for a search algorithm,
 * Contains all methods and attributes that
 * every tree search should have.
 *
 * @author Alked Ejupi Copyright (2019). All rights reserved.
 */
public abstract class TreeSearch {

    int nodesGenerated;
    int depth;

    protected long start;
    protected long end;

    private List<Node> solution;
    private StringBuilder solutionMoves;
    private StringBuilder solutionASCII;

    protected abstract List<Node> treeSearch(Problem problem);

    TreeSearch(){
        this.nodesGenerated = 0;
        this.depth = 0;
        this.solutionASCII = new StringBuilder();
        this.solutionMoves = new StringBuilder();
    }

    public int getNodesGenerated() {
        return nodesGenerated;
    }

    // this is usually used to create the root
    protected Node makeNode(Problem problem, State initialState, boolean heuristic){
        nodesGenerated++;
        return new Node(problem, initialState, heuristic);
    }
    private Node generateChildNode(Problem problem, Node parent, Action action) throws
         IllegalMoveException {
        return new Node(problem, parent, action, false);
    }

    private Node generateHeuristicChildNode(Problem problem, Node parent, Action action) throws
         IllegalMoveException {
        return new Node(problem, parent, action, true);
    }

    protected List<Node> generateRandomSuccessors(Node nodeToExpand, Problem problem){
        List<Node> random = generateSuccessors(nodeToExpand, problem, false);
        Debug.showShuffling();
        Collections.shuffle(random);
        return random;
    }

    protected List<Node> generateSuccessors(Node nodeToExpand, Problem problem, boolean heuristic){
        ArrayList<Node> successors = new ArrayList<Node>();

        Debug.showStartExpansion(nodeToExpand.state);

        for (Action action: problem.actions()) {
            Node child;
            try {
                if(!heuristic){
                    child = generateChildNode(problem, nodeToExpand, action);
                }else{
                    child = generateHeuristicChildNode(problem, nodeToExpand, action);
                }
            } catch (IllegalMoveException e) {
                child = null;
            }

            if(child != null) {
                successors.add(child);
```

```java
                Debug.showChildGenerated(child);
                nodesGenerated++;
            }
        }

        Debug.showEndExpansion(nodeToExpand.state, successors);

        return successors;
    }

    public String solveProblem(Problem problem){
        start = System.currentTimeMillis();
        this.solution = treeSearch(problem);
        end = System.currentTimeMillis();
        return Utils.solutionToString(this);
    }

    /**
     * Return the solution given a node
     * @param solution
     * @return the reconstructed path
     */
    protected List<Node> solution(Node solution) {
        LinkedList<Node> path = new LinkedList<>();

        path.add(solution);
        while (solution.parent != null) {
            path.add(solution.parent);
            solution = solution.parent;
        }

        Collections.reverse(path);
        path.remove(0);

        depth = path.size();
        return path;
    }

    public List<Node> getSolution() {
        return solution;
    }

    public int getDepth() {
        return depth;
    }

    public StringBuilder getSolutionMoves() {
        return solutionMoves;
    }

    public StringBuilder getSolutionASCII() {
        return solutionASCII;
    }

    public long time(){
        return end - start;
    }
}
```

```java
package TreeSearchAlgorithm;

import Exceptions.IllegalMoveException;
import Problem.Problem;
import Problem.TransitionModel.Action;
import BlocksWorld.Board;
import Problem.State;

public class Node implements Heuristic {

    State state;
    Node parent;
    Action action;
    int depth;
    int pathCost;
    int estimatedCost;
```

```java

    //used to define whether a node is heuristic or not
    boolean heuristic;

    /*the root*/
    Node(Problem problem, State start, boolean heuristic){
        this.state = start;
        this.parent = null;
        this.heuristic = heuristic;
        this.depth = 0;
        if(heuristic){
            this.pathCost = 0;
            this.estimatedCost = calculateEstimatedCost(g(), h(problem.goal()));
        }
    }

    /* child (successor) */
    Node(Problem problem, Node parent, Action action, boolean heuristic) throws IllegalMoveException {
        this.state = problem.generateState(parent.state, action);
        this.action = action;
        this.parent = parent;
        this.heuristic = heuristic;
        this.depth = parent.depth + 1;

        if(heuristic){
            this.pathCost = parent.pathCost + problem.actionCost();
            this.estimatedCost = calculateEstimatedCost(g(), h(problem.goal()));
        }
    }

    @Override
    public int g() {
        return pathCost;
    }

    public boolean isHeuristic() {
        return heuristic;
    }

    @Override
    public int h(Board boardGoal) {
        int sum = 0;
        Board boardCurr = state.getBoard();

        int a = boardCurr.getA().manhattanDistance(boardGoal.getA());
        int b = boardCurr.getB().manhattanDistance(boardGoal.getB());
        int c = boardCurr.getC().manhattanDistance(boardGoal.getC());

        int max = Math.max(a, Math.max(b, c));


        if(!(boardCurr.getA().manhattanDistance(boardGoal.getA()) == 0)){
            sum += boardCurr.getA().manhattanDistance(boardGoal.getA());
            if(!((boardCurr.getAgent().manhattanDistance(boardCurr.getA())) > 5)){
                sum += (boardCurr.getAgent().manhattanDistance(boardCurr.getA()));
            }
        }

        if(!(boardCurr.getB().manhattanDistance(boardGoal.getB()) == 0)){
            sum += boardCurr.getB().manhattanDistance(boardGoal.getB());
            if(!((boardCurr.getAgent().manhattanDistance(boardCurr.getB())) > 5)){
                sum += (boardCurr.getAgent().manhattanDistance(boardCurr.getB()));
            }
        }

        if(!(boardCurr.getC().manhattanDistance(boardGoal.getC()) == 0)){
            sum += boardCurr.getC().manhattanDistance(boardGoal.getC());
            if(!((boardCurr.getAgent().manhattanDistance(boardCurr.getC())) > 5)){
                sum += (boardCurr.getAgent().manhattanDistance(boardCurr.getC()));
            }
        }

        return sum + max*4;
    }

```

```java

    public int hOld(Board boardGoal) {
        int sum = 0;
        Board boardCurr = state.getBoard();

        int a = boardCurr.getA().manhattanDistance(boardGoal.getA());
        int b = boardCurr.getB().manhattanDistance(boardGoal.getB());
        int c = boardCurr.getC().manhattanDistance(boardGoal.getC());

        sum = a + b + c;

        return sum;
    }

    @Override
    public int calculateEstimatedCost(int g, int h) {
        return g + h;
    }

    public Action getAction() {
        return action;
    }

    public int getEstimatedCost() {
        return estimatedCost;
    }

    public State getState() {
        return state;
    }

    @Override
    public boolean equals(Object obj) {
        if(obj instanceof Node){
            return toString().equals(obj.toString());
        }
        return false;
    }

    @Override
    public int hashCode() {
        return this.state.hashCode();
    }

    @Override
    public String toString() {
        return state.toString();
    }
}
```

```java
package TreeSearchAlgorithm;

import BlocksWorld.Board;
import BlocksWorld.Cell;

public interface Heuristic {

    /**
     * Path cost from start node to {@code n}
     */
    int g();

    /**
     * estimated cost of the cheapest path from {@code n} to goal
     */
    int h(Board goal);


    /**
     *  estimated cost of the cheapest solution through {@code n}
     */
    int calculateEstimatedCost(int g, int h);
}
```

```java
package TreeSearchAlgorithm;

import Problem.Problem;
import Utils.Debug;
import java.util.*;

public class BFS extends TreeSearch {

    //Uses a FIFO queue to store the fringe
    Queue<Node> fringe;

    public BFS(){
        super();
        this.fringe = new LinkedList<>();
    }

    @Override
    protected List<Node> treeSearch(Problem problem) {
        Node root = makeNode(problem, problem.startState(), false);

        fringe.add(root);
        Debug.showAddingRoot(root);

        while(!fringe.isEmpty()){
            Node nodeToExpand = fringe.remove();

            Debug.showRemoveNodeFromFringe(nodeToExpand);
            Debug.showCheckGoal(nodeToExpand);

            if(problem.checkGoal(nodeToExpand)) {
                Debug.showGoal(nodeToExpand);
                return solution(nodeToExpand);
            }
            Debug.showIsNotGoal(nodeToExpand);

            List<Node> successors = generateSuccessors(nodeToExpand, problem, false);
            fringe.addAll(successors);
            Debug.showAddAllSuccessors(successors.size());
            Debug.showFringe(fringe);
        }
        return null;
    }

}
```

```java

package TreeSearchAlgorithm;

import Problem.Problem;
import Utils.Debug;
import java.util.Stack;
import java.util.List;

public class DFS extends TreeSearch {

    //LIFO queue
    Stack<Node> fringe;

    public DFS(){
        super();
        this.fringe = new Stack<>();
    }

    @Override
    protected List<Node> treeSearch(Problem problem) {
        Node root = makeNode(problem, problem.startState(), false);

        fringe.add(root);
        Debug.showAddingRoot(root);
        Debug.showFringe(fringe);

        while (!fringe.isEmpty()){
            Node nodeToExpand = fringe.pop();

```

```
            Debug.showRemoveNodeFromFringe(nodeToExpand);

            Debug.showCheckGoal(nodeToExpand);
            if(problem.checkGoal(nodeToExpand)) {
                Debug.showGoal(nodeToExpand);
                return solution(nodeToExpand);
            }
            Debug.showIsNotGoal(nodeToExpand);
            List<Node> successors = generateRandomSuccessors(nodeToExpand, problem);
            Debug.showAddAllSuccessors(successors.size());
            fringe.addAll(successors);
            Debug.showFringe(fringe);

        }
        return null;
    }
}
```

```
package TreeSearchAlgorithm;

import Problem.Problem;
import Utils.Debug;
import java.util.List;

public class IDS extends TreeSearch {
    //Finite limit of IDS
    int LIMIT = Integer.MAX_VALUE;

    @Override
    protected List<Node> treeSearch(Problem problem) {

        Node root = makeNode(problem, problem.startState(), false);
        Debug.creatingRoot(root.state.getBoard().getConfiguration());

        Node solution;

        for (int depth = 0; depth < LIMIT; depth++) {
            nodesGenerated = 1;
            Debug.showLimitIteration(depth);
            Debug.showStartDLSCall(root, depth);
            solution = DLS(problem, root, depth);

            Debug.showEndDLSCalls(depth);

            if (solution != null) {
                Debug.showSolutionFoundDLS(depth);
                return solution(solution);
            }
            Debug.showNoSolutionFoundDLS(depth);

        }
        return null;
    }


    private Node DLS(Problem problem, Node current, int depth) {

        Debug.showCheckGoal(current);
        if (depth == 0 && problem.checkGoal(current)) {
            Debug.showGoalDFS(current);
            return current;
        }

        Debug.showIsNotGoal(current);

        if (depth > 0) {
            List<Node> successors = generateSuccessors(current, problem, false);

            Debug.showDFSCallOnSuccessors(successors);
            for(Node successor : successors) {
                Debug.showCallDLSOnChild(successor);
                Node result = DLS(problem, successor, depth - 1);
                if (result != null) {
                    return result;
                }
```

```
            }
        }
        return null;
    }
}
```

```
package TreeSearchAlgorithm;

import Problem.Problem;
import Utils.Debug;
import java.util.Comparator;
import java.util.List;
import java.util.PriorityQueue;


public class AStar extends TreeSearch {

    //Uses a priority queue to store the fringe
    PriorityQueue<Node> fringe;

    public AStar(){
        super();          //Comparator that compares nodesGenerated using their estimated cost
        this.fringe = new PriorityQueue<>(Comparator.comparingInt(node -> node.estimatedCost));
    }

    @Override
    protected List<Node> treeSearch(Problem problem) {
        Node root = makeNode(problem, problem.startState(), true);

        fringe.add(root);
        Debug.showAddingRoot(root);
        Debug.showFringe(fringe);

        while(!fringe.isEmpty()){
            Node nodeToExpand = fringe.remove();
            Debug.showRemoveNodeFromFringe(nodeToExpand);
            Debug.showFringe(fringe);

            Debug.showCheckGoal(nodeToExpand);
            if(problem.checkGoal(nodeToExpand)) {
                Debug.showGoal(nodeToExpand);
                return solution(nodeToExpand);
            }
            Debug.showIsNotGoal(nodeToExpand);

            List<Node> successors = generateSuccessors(nodeToExpand, problem, true);

            fringe.addAll(successors);
            Debug.showAddAllSuccessors(successors.size());
            Debug.showFringe(fringe);

        }
        return null;
    }
}
```

B.2 `Problem` package

```
package Problem;

import BlocksWorld.Cell;
import Exceptions.IllegalMoveException;
import BlocksWorld.Board;
import Problem.TransitionModel.Action;
import TreeSearchAlgorithm.Node;

import java.awt.*;

import static BlocksWorld.Cell.*;
import static BlocksWorld.Cell.CellType.EMPTY;
import static Utils.Utils.*;

```

```java
/**
 * This class defines the problem to solve.
 *
 * @author Alked Ejupi Copyright (2019). All rights reserved.
 */

public class BlocksWorldTileProblem implements Problem {

    private static int N = 4;
    private static int STEP_COST = 1;

    public TransitionModel transitionModel;

    public String goalConfiguration;

    public Board goalBoard;
    public State startState;

    private static CellType A = CellType.A;
    private static CellType B = CellType.B;
    private static CellType C = CellType.C;
    private static CellType AGENT = CellType.AGENT;

    public BlocksWorldTileProblem(Point...points){
        this.transitionModel = transitionModel();
        this.startState = initStart(points[0], points[1], points[2], points[3]);
        this.goalConfiguration = initGoal(points[4], points[5], points[6], points[7]);
    }

    public BlocksWorldTileProblem(String startConfiguration, String goalConfiguration){
        this.transitionModel = transitionModel();
        this.startState = initStart(startConfiguration);
        this.goalConfiguration = initGoal(goalConfiguration);
    }

    public State initStart(Point...points){
        Cell a = new Cell(points[0], A);
        Cell b = new Cell(points[1], B);
        Cell c = new Cell(points[2], C);
        Cell ag = new Cell(points[3], AGENT);
        Cell[][] cells = generateGridCells(a, b, c, ag, N);
        Board board = generateBoard(cells, N);
        return new State(board);
    }

    public String initGoal(Point...points){
        Cell a = new Cell(points[0], A);
        Cell b = new Cell(points[1], B);
        Cell c = new Cell(points[2], C);
        Cell ag = new Cell(points[3], AGENT);
        Cell[][] cells = generateGridCells(a, b, c, ag, N);
        this.goalBoard = generateBoard(cells, N);
        return goalBoard.getConfiguration();
    }

    public State initStart(String startConfiguration){
        Cell[][] cells = convert1DTo2DCells(convertStringTo1DCells(startConfiguration));
        Board board = generateBoard(cells, N);
        return new State(board);
    }

    public String initGoal(String goalConfiguration){
        Cell[][] cells = convert1DTo2DCells(convertStringTo1DCells(goalConfiguration));
        this.goalBoard = generateBoard(cells, N);
        return goalBoard.getConfiguration();
    }

    @Override
    public TransitionModel transitionModel() {
        return new TransitionModel();
    }

    @Override
    public Action[] actions() {
        return Action.values();
    }
```

```java

    @Override
    public State startState() {
        return startState;
    }

    @Override
    public Board goal() {
        return goalBoard;
    }

    public void setGoal(String goal) {
        this.goalConfiguration = goal;
    }

    @Override
    public State generateState(State parent, Action action) throws IllegalMoveException {
        Board parentBoard = parent.getBoard();
        // new cells for A, B, C and agent
        Cell An = cloneCell(parentBoard.getA());
        Cell Bn = cloneCell(parentBoard.getB());
        Cell Cn = cloneCell(parentBoard.getC());
        Cell AGn = cloneCell(parentBoard.getAgent());
        //new cells
        Cell[][] cells = generateGridCells(An, Bn, Cn, AGn, parentBoard.getN());
        //new Board
        Board board = generateBoard(cells, parentBoard.getN());
        //new State
        State newState = new State(board);
        transitionModel.performTransition(action, newState);
        return newState;
    }

    public String getGoalConfiguration() {
        return goalConfiguration;
    }

    @Override
    public int actionCost() {
        return STEP_COST;
    }

    public Board getGoalBoard() {
        return goalBoard;
    }

    @Override
    public boolean checkGoal(Node solution) {
        State state = solution.getState();
        return getGoalConfiguration().replace(AGENT.getText(), EMPTY.getText())
                .equals(state.getBoard().getConfiguration().replace(AGENT.getText(), EMPTY.getText()));
    }

}
```

```java
package Problem;

import Exceptions.IllegalMoveException;
import Problem.TransitionModel.Action;
import BlocksWorld.Board;
import TreeSearchAlgorithm.Node;

/**
 * Interface for defining the Blocks World Tile puzzle problem
 *
 * @author Alked Ejupi. Copyright (2019). All rights reserved.
 *
 */
public interface Problem {

    TransitionModel transitionModel();

    Action[] actions();

    State startState();
```

```java

    Board goal();

    State generateState(State parentState, Action action) throws IllegalMoveException;

    int actionCost();

    /**
     * Checks by just comparing
     * the String configuration (of the Board)
     *
     * Replaces the string of the agent with
     * the string that represents an empty space
     * since the position of the agent does not matter
     * when checking the goal state
     * @param solution the node to check
     * @return true if it is the goal
     */
    boolean checkGoal(Node solution);
}
```

```java
package Problem;

import BlocksWorld.Board;
import Problem.TransitionModel.Action;

/**
 * Class that represent a single state of the puzzle,
 * in other words the arrangements of the pieces.
 * It also provides the possible action that can be performed.
 *
 * @author Alked Ejupi Copyright (2019). All rights reserved.
 */
public class State {

    Board board;
    Action actionTaken;


    public State(Board board) {
        this.board = board;
    }

    public Board getBoard() {
        return board;
    }

    @Override
    public boolean equals(Object obj) {
        if(obj instanceof State){
            return board.equals(((State) obj).board);
        }
        return super.equals(obj);
    }

    public String ascii(){
        return board.getASCIIString();
    }

    public Action getActionTaken() {
        return actionTaken;
    }

    public void setActionTaken(Action actionTaken) {
        this.actionTaken = actionTaken;
    }

    @Override
    public String toString() {
        return board.toString();
    }

    @Override
    public int hashCode() {
        return board.hashCode();
```

```
    }

}
```

```
package Problem;

import Exceptions.IllegalMoveException;
import BlocksWorld.Board;
import BlocksWorld.Cell;
import java.awt.Point;

/**
 * Represents all possible actions
 * that can be taken in puzzle game
 *
 *  @author Alked Ejupi Copyright (2019). All rights reserved.
 */

public class TransitionModel {

    /* Enum class for agent moves */
    public enum Action {UP, DOWN, LEFT, RIGHT}

    public void performTransition(Action move, State state) throws IllegalMoveException {
        Board board = state.getBoard();
        Cell destination = null;

        switch (move){
            case UP:  destination = up(board); break;
            case DOWN: destination = down(board); break;
            case LEFT: destination = left(board); break;
            case RIGHT:destination = right(board); break;
        }

        if(destination == null){
            throw new IllegalMoveException(move);
        }else{
            moveAgent(board, destination);
            state.setActionTaken(move);
            //update configuration
            board.updateConfiguration();
        }
    }

    private void moveAgent(Board board, Cell destination){

        Cell[][] cells = board.getCells();
        Point agent = board.getAgent().getPoint();

        Point tempAgent = new Point(((int) agent.getX()), ((int) agent.getY()));
        Point tempDestination = new Point(((int) destination.getX()), ((int) destination.getY()));

        Cell temp = cells[agent.x][agent.y];
        cells[agent.x][agent.y] = cells[destination.x][destination.y];
        cells[agent.x][agent.y].setLocation(tempAgent);

        cells[tempDestination.x][tempDestination.y] = temp;
        cells[tempDestination.x][tempDestination.y].setLocation(tempDestination);

    }

    private Cell left(Board board) {
        Cell agent = board.getAgent(); // remember to reset
        int newY = (int) (agent.getY() - 1);
        Point left = new Point(agent.x, newY);

        Cell destination;

        try {
            destination = board.getCells()[left.x][left.y];
        }catch (ArrayIndexOutOfBoundsException e){
            return null;
        }

        return destination;
```

```java

    }

    private Cell right(Board board) {
        Cell agent = board.getAgent(); // remember to reset
        int newY = (int) (agent.getY() + 1);
        Point right = new Point(agent.x, newY);
        Cell destination;
        try {
            destination = board.getCells()[right.x][right.y];
        }catch (ArrayIndexOutOfBoundsException e){
            return null;
        }
        return destination;
    }

    private Cell down(Board board) {
        Cell agent = board.getAgent(); // remember to reset
        int newX = (int) (agent.getX() + 1);
        Point down = new Point(newX, agent.y);
        Cell destination;
        try {
            destination = board.getCells()[down.x][down.y];
        }catch (ArrayIndexOutOfBoundsException e){
            return null;
        }
        return destination;
    }

    private Cell up(Board board){
        Cell agent = board.getAgent(); // remember to reset
        int newX = (int) (agent.getX() - 1);
        Point up = new Point(newX, agent.y);
        Cell destination;

        try {
            destination = board.getCells()[up.x][up.y];
        }catch (ArrayIndexOutOfBoundsException e){
            return null;
        }

        return destination;
    }
}
```

B.3 BlocksWorld package

```java
package BlocksWorld;

import Utils.Utils;
import static Utils.Utils.*;

/**
 * Represents a board containing NxN cells.
 * It remembers the position of tiles a, b, c and the agent.
 *
 * the {@code configuration} attribute represents the
 * arrangements of tiles and agent as a single {@code String}
 *
 * @author Alked Ejupi Copyright (2019). All rights reserved.
 *
 *
 */
public class Board {
    int N;

    private Cell[][] cells;
    private String configuration;
    Cell a, b, c, agent;

    public Board(int N, Cell[][] cells) {
        this.N = N;
        this.cells = cells;
```

```
        this.configuration = convert1DCellsToString(convert2DCellsTo1DCells(cells));
    }


    public void updateConfiguration(){
        this.configuration = convert1DCellsToString(convert2DCellsTo1DCells(getCells()));
    }


    public String getConfiguration() {
        return configuration;
    }

    public Cell[][] getCells() {
        return cells;
    }

    public int getN() {
        return N;
    }

    public String getASCIIString(){
        return Utils.drawGridCells(Utils.convert2DCellsTo1DCells(getCells()), getN());
    }

    public void setA(Cell a) {
        this.a = a;
    }

    public void setB(Cell b) {
        this.b = b;
    }

    public void setC(Cell c) {
        this.c = c;
    }

    public void setAgent(Cell agent) {
        this.agent = agent;
    }

    public Cell getA() {
        return a;
    }

    public Cell getB() {
        return b;
    }

    public Cell getC() {
        return c;
    }

    public Cell getAgent() {
        return agent;
    }

    @Override
    public boolean equals(Object obj) {
        if(obj instanceof Board){
            return toString().equals(obj.toString());
        }
        return super.equals(obj);
    }

    @Override
    public String toString() {
        return configuration.replace(Cell.CellType.AGENT.getText(), Cell.CellType.EMPTY.getText());
    }

    @Override
    public int hashCode() {
        return toString().hashCode();
    }
}
```

```java
package BlocksWorld;

import java.awt.*;

/**
 * Basic representation of a single cell
 * defining its positions (X, y) and its {@code CellType}
 *
 *  @author Alked Ejupi Copyright (2019). All rights reserved.
 *
 */
public class Cell extends Point implements ManhattanDistance {

    private CellType cellType;

    /**
     * Enum class for representing
     * the cell type.
     */
    public enum CellType {
        A("A"),
        B("B"),
        C("C"),
        AGENT("@"),
        EMPTY("-");

        private String text;

        CellType(String text) {
            this.text = text;
        }
        public String getText() {
            return this.text;
        }
    }

    /**
     * Calculates Manhattan Distance between two cells
     * @param c the cell to calculate the distance
     * @return the manhattan distance
     */
    @Override
    public int manhattanDistance(Cell c) {
        return Math.abs(this.x - c.x) + Math.abs(this.y - c.y);
    }


    public Cell(Point point, CellType cellType){
        super(point);
        this.cellType = cellType;
    }

    public Cell(int x, int y, CellType cellType){
        super(x, y);
        this.cellType = cellType;
    }

    public CellType getCellType() {
        return cellType;
    }

    public Point getPoint() {
        return super.getLocation();
    }

    public void setCellType(CellType cellType) {
        this.cellType = cellType;
    }

}
```

```java
package BlocksWorld;

public interface ManhattanDistance {
```

```
    int manhattanDistance(Cell c);
}
```

B.4 Utils package

```
package Utils;

import BlocksWorld.Board;
import BlocksWorld.Cell;
import Problem.BlocksWorldTileProblem;
import TreeSearchAlgorithm.Node;
import TreeSearchAlgorithm.TreeSearch;

import java.awt.*;

/**
 * Utility class
 *
 * @author Alked Ejupi Copyright (2019). All rights reserved.
 */

public final class Utils {
    /**
     * Draws the blocks of the puzzle in a nice way in the Console.
     *
     * @param array1D the puzzle in 2d array form
     * @param n the size of grid (nxn)
     * @return the String containing the puzzle with blocks
     */
    public static String drawGridCells(Cell[] array1D, int n) {

        String[] cellValues = new String[array1D.length];
        for (int i = 0; i < array1D.length; i++) {
            if(array1D[i].getCellType().getText().equals(Cell.CellType.EMPTY.getText()))
               cellValues[i] = " ";
            else
                cellValues[i] = array1D[i].getCellType().getText();

        }
        String pattern = buildPattern(n);
        String[] R ={"             ","        o    ","             ","             ","             "};
        StringBuilder r = new StringBuilder();

        for (int X: pattern.getBytes()) {
            for (int x: pattern.replace("1",R[X-=48].length()>5?"151":"111").getBytes()){
                r.append(R[X].charAt(x - 48));
            }
            r.append("\n");
        }

        for(String i: cellValues) {
            r = new StringBuilder(r.toString().replaceFirst("o", i.equals("") ? "b" + i : i));
        }
        return r.toString();
    }


    public static Cell[][] convert1DTo2DCells(Cell[] array) {

        int n = (int) Math.sqrt(array.length);

        if (array.length != (n*n))
            throw new IllegalArgumentException("Invalid array length");

        Cell[][] cells = new Cell[n][n];

        for(int x=0; x<n; x++) {
            for (int y = 0; y < n; y++) {
                Cell c = array[(x * n) +y];
                c.setLocation(x, y);
                cells[x][y] = c;
            }
        }
```

```java
        return cells;
    }
    /**
     * Build a pattern depending on the size of the grid
     * @param n the size
     * @return the pattern built
     */
    private static String buildPattern(int n){
        String pattern = "0";

        for (int i = 0; i < n-1; i++) {
            pattern += "12";
        }
        pattern += "14";
        return pattern;
    }

    public static String convert1DCellsToString(Cell[] array1D){
        StringBuilder s = new StringBuilder();
        for (Cell c: array1D) s.append(c.getCellType().getText());
        return s.toString();
    }


    public static Cell[] convertStringTo1DCells(String cells){
        Cell[] cells1 = new Cell[cells.length()];
        for (int i = 0; i < cells1.length; i++) {
            switch (String.valueOf(cells.charAt(i))){
                case "A": cells1[i] = new Cell(0, 1, Cell.CellType.A); break;
                case "B": cells1[i] = new Cell(0, 2, Cell.CellType.B); break;
                case "C": cells1[i] = new Cell(0, 3, Cell.CellType.C); break;
                case "@": cells1[i] = new Cell(0, 4, Cell.CellType.AGENT); break;
                case "-": cells1[i] = new Cell(0, 5, Cell.CellType.EMPTY); break;
            }
        }
        return cells1;
    }


    public static Cell[] convert2DCellsTo1DCells(Cell[][] array2D){
        Cell[] array1D = new Cell[array2D.length*array2D.length];

        for(int i = 0; i < array2D.length; i++) {
            Cell[] row = array2D[i];
            System.arraycopy(array2D[i], 0, array1D, i * row.length, row.length);
        }

        return array1D;
    }


    public static boolean isDebuggerON() {
        return Debug.DEBUGGER;
    }

    public static Board generateBoard(Cell[][] cells, int n){
        Board newBoard = new Board(n, cells);
        for (int x = 0; x < n; x++) {
            for (int y = 0; y < n ; y++) {
                Cell cell = cells[x][y];
                switch (cell.getCellType()) {
                    case A: newBoard.setA(cell); break;
                    case B: newBoard.setB(cell); break;
                    case C: newBoard.setC(cell); break;
                    case AGENT: newBoard.setAgent(cell); break;
                }
            }

        }
        newBoard.updateConfiguration();
        return newBoard;
    }

    public static Cell cloneCell(Cell c){
        return new Cell(((int) c.getX()), ((int) c.getY()), c.getCellType());
    }
```

```

    public static Cell[][] generateGridCells(Cell a, Cell b, Cell c, Cell agent, int n){
        Cell[][] cells = new Cell[n][n];
        for (int x = 0; x < n ; x++) {
            for (int y = 0; y < n; y++) {
                if(x==a.x && y == a.y){
                    cells[x][y] = a;
                }else if(x==b.x && y == b.y){
                    cells[x][y] = b;
                }else if(x==c.x && y == c.y){
                    cells[x][y] = c;
                }else if(x==agent.x && y == agent.y){
                    cells[x][y] = agent;
                }else{
                    cells[x][y] = new Cell(new Point(x,y), Cell.CellType.EMPTY);
                }
            }
        }
        return cells;
    }



    public static void newLine(){
        System.out.println();
    }

    public static void println(String str){
        System.out.println(str);
    }

    public static void printStartAndGoal(BlocksWorldTileProblem problem1) {
        println(" INITIAL STATE");
        println(problem1.startState().ascii());
        println(" GOAL STATE");
        if (problem1.getGoalConfiguration() == null) {
          println(drawGridCells(convert2DCellsTo1DCells(problem1.getGoalBoard().getCells()),
                    problem1.getGoalBoard().getN()));
        }else{
            int n = (int) Math.sqrt(problem1.goalConfiguration.length());
            println(drawGridCells(convertStringTo1DCells(problem1.getGoalConfiguration()), n));
        }

    }

    public static String solutionToString(TreeSearch search) {

        int i = 1;
        for (Node node: search.getSolution()) {
            search.getSolutionMoves().append(node.getAction()).append(" ");
            search.getSolutionASCII().append("Step ")
                    .append(i++)
                    .append(": ")
                    .append(node.getState().getActionTaken())
                    .append("\n")
                    .append("Configuration: ")
                    .append(node.getState().getBoard().getConfiguration())
                    .append("\n")
                    .append(node.getState().ascii())
                    .append("\n");
        }

        return search.getSolutionASCII() +
                    "\nTime elapsed: " + search.time() + "ms" +
                    "\nNumber nodes generated: " + search.getNodesGenerated() +
                    "\nDepth solution : " + search.getDepth() +
                    "\nMoves: " + search.getSolutionMoves();
    }
}
```

```
package Utils;

import Problem.State;
import TreeSearchAlgorithm.Node;

```

```java
import java.util.Collection;
import java.util.Iterator;
import java.util.List;
import java.util.PriorityQueue;

import static Utils.Utils.newLine;
import static Utils.Utils.println;

/**
 * Debug class
 *
 * @author Alked Ejupi Copyright (2019). All rights reserved.
 */


public class Debug {

    public static boolean ON = true;
    public static boolean OFF = false;

    public static boolean DEBUGGER;

    public static void setDEBUGGER(boolean DEBUGGER) {
        Debug.DEBUGGER = DEBUGGER;
    }

    public static void showRemoveNodeFromFringe(Node nodeToRemove){

        if(DEBUGGER){
            println("Removing node " + nodeToRemove.getState().getBoard().getConfiguration() + " from the fringe");
            println(" ");
        }
    }

    public static void showHeuristicFringe(Collection<Node> fringe){
        if(DEBUGGER){
            println("    FRINGE    ");
            println("                                                 ");

            Iterator<Node> it = fringe.iterator();

            int j = 1;
            if(fringe.size() == 0){
                println("      empty        ");
            }else{
                while (it.hasNext()) {
                    Node node = it.next();
                    println("   "+node.getState().getBoard().getConfiguration() + ""+" (cost=" + node.getEstimatedCost() + ")");
                }
            }
            println("                                                 ");
        }

    }

    public static void showFringe(Collection<Node> fringe){
        if(DEBUGGER) {
            if (fringe instanceof PriorityQueue) {
                showHeuristicFringe(fringe);
            } else {
                println("    FRINGE    ");
                println("                                                 ");

                Iterator<Node> it = fringe.iterator();

                int j = 1;
                if (fringe.size() == 0) {
                    println("      empty        ");
                } else {
                    while (it.hasNext()) {
                        Node node = it.next();
                        println("  " + node.getState().getBoard().getConfiguration()
                                + "   " + " (" + j++ + ")");
                    }
```

```
                }
                println("                                                     ");
            }
        }

    }

    public static void showCheckGoal(Node nodeToExpand) {
        if(DEBUGGER)
            println("Check if " + nodeToExpand.getState().getBoard().getConfiguration() + " is the goal...");
    }

    public static void showAddingRoot(Node node) {
        if(DEBUGGER) {
            println("Adding root " + node.getState().getBoard().getConfiguration() + "to the fringe.");

        }
    }

    public static void showGoal(Node nodeGoal) {
        if(DEBUGGER){
            println("Node " + nodeGoal.getState().getBoard().getConfiguration() + " is the goal!");
            println(" ");
            println("Solution:");
        }
    }

    public static void showIsNotGoal(Node node) {
        if(DEBUGGER)
            println("It is not the goal, " + " then expand node "+ node.getState().getBoard().getConfiguration());
    }

    public static void showStartExpansion(State state) {
        if(DEBUGGER){
            Utils.newLine();
            println("Start expanding node " + state.getBoard().getConfiguration());
            Utils.newLine();
        }

    }

    public static void showAddAllSuccessors(int size) {
        if(DEBUGGER){
            println("Adding " + size + " successors to the fringe.");
        }
    }

    public static void showChildGenerated(Node child) {
        if(DEBUGGER){
            println("Child: " + child.getState().getBoard().getConfiguration());
            if(child.isHeuristic()){
                println("estimated cost : "+ child.getEstimatedCost());
            }
            println("Action taken: " + child.getState().getActionTaken().name());
            println(child.getState().ascii());
            Utils.newLine();
        }
    }

    public static void showEndExpansion(State state, List<Node> successors) {
        if(DEBUGGER){
            println("End expansion of " + state.getBoard().getConfiguration());
            println("No. successors generated: " + successors.size());
            Utils.newLine();
        }
    }


    public static void showShuffling() {
        if(DEBUGGER){
            println("Shuffling the order of the successors...");
        }
    }

```

```
    public static void showLimitIteration(int depth) {
        if(DEBUGGER){
            newLine();
            println("Performing IDS at depth limit " + depth);
            newLine();
        }
    }

    public static void showStartDLSCall(Node node, int depth) {
        if(DEBUGGER){
            println("Performing recursive DLS calls (d=" + depth +") on root node "+
                node.getState().getBoard().getConfiguration());
        }
    }

    public static void showEndDLSCalls(int depth) {
        if(DEBUGGER){
            newLine();
            println("End of recursive DLS calls with (d=" + depth +")");
            newLine();
        }
    }

    public static void showSolutionFoundDLS(int depth) {

        if(DEBUGGER){
            println("Solution found at (d=" + depth +")");
        }
    }

    public static void showNoSolutionFoundDLS(int depth) {
        if(DEBUGGER){
            println("Solution not found at depth limit "+ depth);
            println("End performing IDS at depth limit "+ depth);
        }
    }

    public static void showCallDLSOnChild(Node state) {
        if(DEBUGGER){
            newLine();
            println("Performing recursive DLS on " + state.getState().getBoard().getConfiguration());
        }
    }

    public static void showGoalDFS(Node current) {
        if(DEBUGGER){
            println("Ending recursive DLS at " + current.getState().getBoard().getConfiguration());
            println("Solution found, return value to IDS.");
        }
    }

    public static void creatingRoot(String config) {
        if(DEBUGGER){
            println("Creating root with initial state: " + config);
            newLine();
        }
    }

    public static void showDFSCallOnSuccessors(List<Node> successors) {
        if(DEBUGGER){
            println("Calling DFS on " + successors.size() + " successors");
        }
    }



}
```

B.5 Exceptions **package**

```
package Exceptions;

import Problem.TransitionModel.Action;
```

```java
import java.awt.*;

/**
 * Exception for an illegal move of the agent.
 */
public class IllegalMoveException extends Exception {

    private Action a;

    public IllegalMoveException(Action a){
        super();
        this.a = a;
    }

    @Override
    public String getMessage() {
        return "["+a+"]"+" is an illegal move.";
    }
}
```

B.6 Main.java

```java
import Problem.BlocksWorldTileProblem;
import Problem.Problem;
import TreeSearchAlgorithm.*;
import Utils.Debug;

import java.awt.Point;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

import static Utils.Debug.*;
import static Utils.Utils.*;

/**
 * Run the algorithms here.
 *
 * the BlocksWorldTileProblem can be defined through 4 points (A, B, C and Agent)
 * or via a configuration String.
 *
 * @author Alked Ejupi Copyright (2019). All rights reserved.
 */
public class Main {



    public static void main(String[] args){

        Point a = new Point(3,0);
        Point b = new Point(3,1);
        Point c = new Point(3,2);
        Point agent = new Point(3,3);

        Point aGoal = new Point(1,1);
        Point bGoal = new Point(2,1);
        Point cGoal = new Point(3,1);
        Point agentGoal = new Point(2,0);

        BlocksWorldTileProblem problem1 = new BlocksWorldTileProblem(a, b, c, agent, aGoal, bGoal, cGoal,
            agentGoal);

        // argument
        String algorithm = args[0];

        Debug.setDEBUGGER(ON);

        BlocksWorldTileProblem evidenceDepth1 =
                new BlocksWorldTileProblem("------------ABC@","------------AB@C");

        BlocksWorldTileProblem evidenceDepth2 =
                new BlocksWorldTileProblem("------------ABC@","------------A@BC");

```

```java

        solveUserProblem(evidenceDepth2, algorithm);
        // solveDifferentPuzzle(problems(), algorithm);


    }



    private static void solveDifferentPuzzle(List<BlocksWorldTileProblem> problems, String algorithm) {
        int i = 1;
        for (BlocksWorldTileProblem problem: problems) {
            if(i > 0) {
                println(" ");
                println("I'm solving the puzzle with " + algorithm + "...");
                println("Problem: " + i++);
                printStartAndGoal(problem);
                solvePuzzle(problem, algorithm);
            }else
                i++;
        }
    }

    private static void solveUserProblem(BlocksWorldTileProblem problem, String algorithm) {
        printStartAndGoal(problem);
        println("I'm solving the puzzle with "+ algorithm + "...");
        solvePuzzle(problem, algorithm);
    }


    /**
     * Solve problem.
     *
     * @param problem the problem
     * @param algorithm the algorithm
     */
    public static void solvePuzzle(Problem problem, String algorithm){

        String output = null;
            switch (algorithm) {
                case "BFS":
                    TreeSearch bfs = new BFS();
                    output = bfs.solveProblem(problem);
                    break;
                case "DFS":
                    TreeSearch dfs = new DFS();
                    output = dfs.solveProblem(problem);
                    break;
                case "IDS":
                    TreeSearch ids = new IDS();
                    output = ids.solveProblem(problem);
                    break;
                case "AStar":
                    TreeSearch ashs = new AStar();
                    output = ashs.solveProblem(problem);

            }

            println(output);
    }




    private static List<BlocksWorldTileProblem> problemsForScalability(){
        List<BlocksWorldTileProblem> problems = new ArrayList<>();

        String start = "------------ABC@";

        problems.add(new BlocksWorldTileProblem(start,"------------AB@C"));
        problems.add(new BlocksWorldTileProblem(start,"------------A@BC"));
        problems.add(new BlocksWorldTileProblem(start,"----------C-AB@-"));
        problems.add(new BlocksWorldTileProblem(start,"----------C-A@B-"));
        problems.add(new BlocksWorldTileProblem(start,"----------B-A-@C"));
        problems.add(new BlocksWorldTileProblem(start,"---------C@-A-B-"));
        problems.add(new BlocksWorldTileProblem(start,"--------A---BC@-"));
```

```java
        problems.add(new BlocksWorldTileProblem(start,"--------BA---@C-"));
        problems.add(new BlocksWorldTileProblem(start,"--------BA---C@-"));
        problems.add(new BlocksWorldTileProblem(start,"--------AB---@-C"));
        problems.add(new BlocksWorldTileProblem(start,"----------C@-A-B"));
        problems.add(new BlocksWorldTileProblem(start,"--------BC--A@--"));
        problems.add(new BlocksWorldTileProblem(start,"-----C---A---B@-"));
        problems.add(new BlocksWorldTileProblem(start,"--------AC@B----"));
        problems.add(new BlocksWorldTileProblem(start,"----A---@C---B--"));

        return problems;

    }

}
```

References

[1] Peter Norvig Stuart Russel. *Artificial Intelligence - A modern approach (3rd edition)*. Prentice Hall Press Upper Saddle River, NJ, USA, 2009.