COMP2208 - Search Methods Coursework

Daniel Best (Student ID: 29777127)

November 28, 2019

1 Approach

My approach to this assignment was to first create support classes that would handle any details that didn't directly relate to the search algorithm itself. These are as follows:

1.1 Node

This class defines both a singular node in the tree structure, as well as the entire tree structure itself by means of its **parent** and **children** variables. It contains a single **Grid** object, the **value** variable. It also implements the Comparable interface, where it compares an optional **estimatedCost** variable - a feature that was specifically added for the A* Search algorithm.

1.2 Grid

A class that handles the state by means of manipulating a **char[][]** multidimensional array. This class stores the actual state of the problem, storing the location of the **agent** (¬) and all of the non-white space blocks; it also allows for that state to be manipulated in a multitude of ways:

- Generates the start and solution state.
- Moves the agent in the grid, thereby changing the location of both the agent and the block it moves to.
- Calculates the **Manhattan distance** between the Grid and another Grid object passed to it, which is used as the heuristic for A* Search.

1.3 Search

An abstract class that defines a common start and solution state for each of the individual search methods, and provides a common **expandNode()** method, which generates the children of a given node.

Using this **Search** class, I was able to easily implement a class for each of the four search algorithms:

1.4 Breadth First Search (BFS)

Uses a **Queue** to store expanded nodes, meaning nodes are checked in the order they are expanded.

1.5 Depth First Search (DFS)

Uses a **Stack** to store expanded nodes, meaning the last node that was expanded is checked next.

1.6 Iterative Deepening Search (IDS)

Uses **Depth Limited Search (DLS)**, a modified version of **DFS** that does not expand nodes at a given **depth**. The main algorithm then iteratively increases this limit.

1.7 A* Heuristic Search

Makes use of an evaluation function to determine which node to pick next, i.e. the **depth** of the node plus the **Manhattan distance (heuristic)** to the solution.

2 Evidence of Search Methods

When testing my search methods, I found 21 start states (as seen in appendix A) that each corresponded to a problem which has an optimal solution at a different depth (from 0 to 20). The solution state was always the same, as specified by the problem in the assignment briefing.

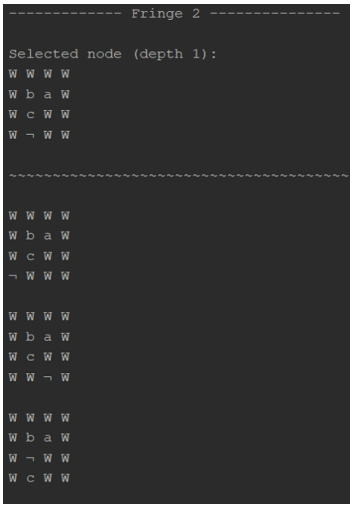
2.1 Breadth First Search (BFS)

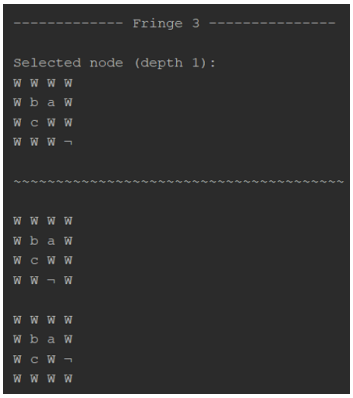
Example 1 (depth 4):

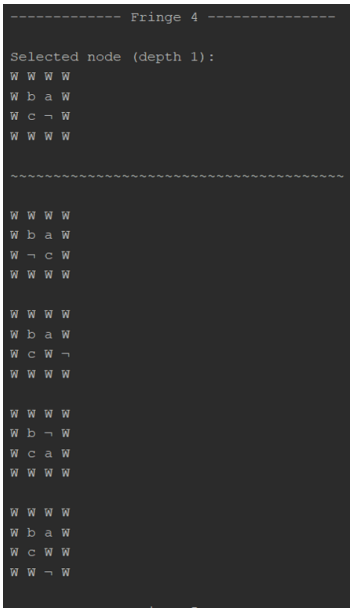
In this initial state, the agent has three possible directions that it can move.

```
********************************************************************************
Breadth First Search (BFS) - Optimal Solution Depth: 4
********************************************************************************

Root:
W W W W
W b a W
W c W W
W W ¬ W

------------- Fringe 1 ---------------

Selected node (depth 0):
W W W W
W b a W
W c W W
W W ¬ W

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

W W W W
W b a W
W c W W
W ¬ W W

W W W W
W b a W
W c W W
W W W ¬

W W W W
W b a W
W c ¬ W
W W W W
```

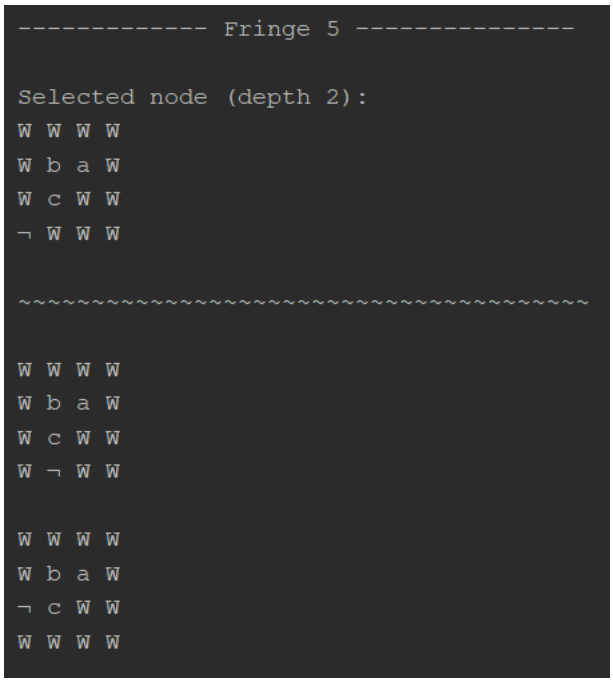
These three new nodes are then expanded in the order that they were first generated.



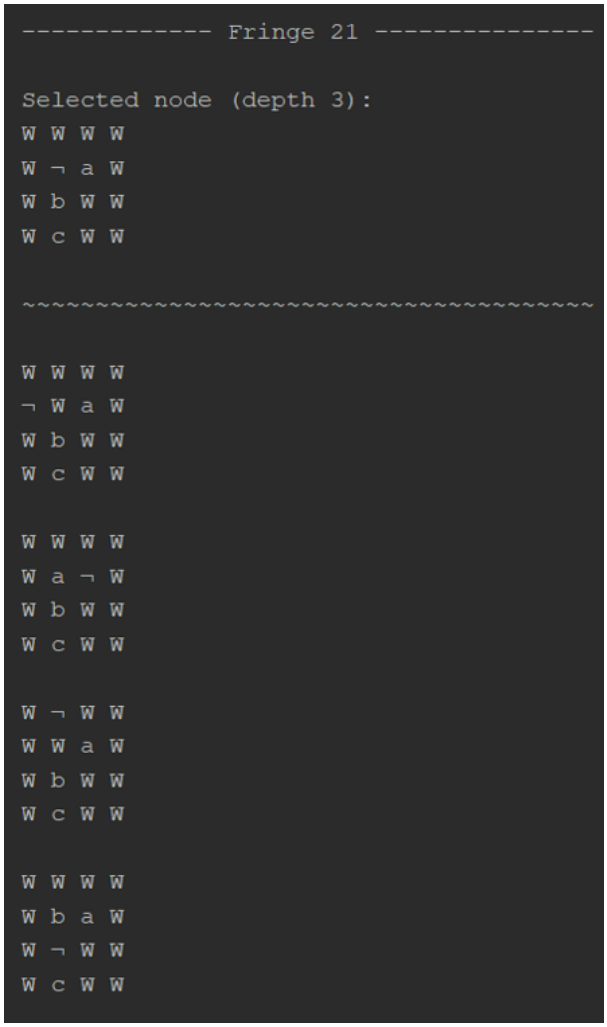




The algorithm then expands the first child node in **Fringe 2**, which generates the first nodes that are at a depth of 2.



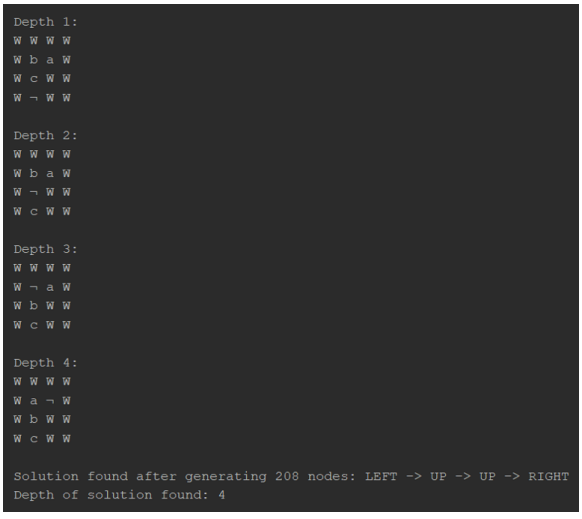
This process then repeats indefinitely until the correct node is found. At the 21st expansion to the fringe, we can see the correct node being found.



However, due to the uninformed nature of the algorithm, it does not recognise this as it only checks if it is in the goal state when taking it off the **Queue**. It therefore continues to go through each node on depth 3, and all the nodes in depth 4 that existed prior to that node - a further 44 expansions.

Fringe 65 shows the node that was generated prior to the goal state node being expanded.

The following is a breakdown of the solution of this problem, as retrieved by this search.



Example 2 (depth 14):

The original problem provided for the assignment has an optimal solution at **depth 14**, but running BFS on it causes a **Java Heap space error** due to the number of generated nodes (space complexity).

The algorithm starts in the same way that it did for the previous problem, by expanding the root node, and then by expanding each subsequent child in the order that they were generated.

```
********************************************************************************
Breadth First Search (BFS) - Optimal Solution Depth: 14
********************************************************************************

Root:
W W W W
W W W W
W W W W
a b c ¬

------------- Fringe 1 ---------------

Selected node (depth 0):
W W W W
W W W W
W W W W
a b c ¬

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

W W W W
W W W W
W W W W
a b ¬ c

W W W W
W W W W
W W W ¬
a b c W
```

```
------------- Fringe 2 ---------------

Selected node (depth 1):
W W W W
W W W W
W W W W
a b ¬ c

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

W W W W
W W W W
W W W W
a ¬ b c

W W W W
W W W W
W W W W
a b c ¬

W W W W
W W W W
W W ¬ W
a b W c
```

```
------------- Fringe 3 ---------------

Selected node (depth 1):
W W W W
W W W W
W W W ¬
a b c W

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

W W W W
W W W W
W W ¬ W
a b c W

W W W W
W W W ¬
W W W W
a b c W

W W W W
W W W W
W W W W
a b c ¬
```

It then continues this process of first checking all of the nodes on a depth in the order they were generated, and then moving down to the next depth.

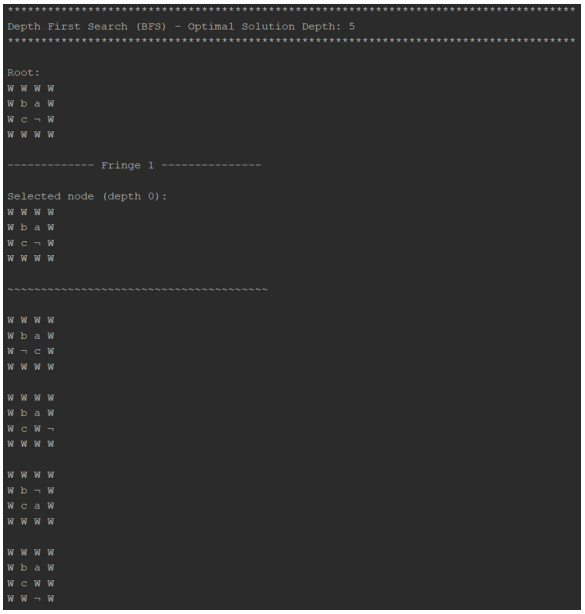
Eventually however, it will run into an **OutOfMemoryException** due to running out of Java Heap space.

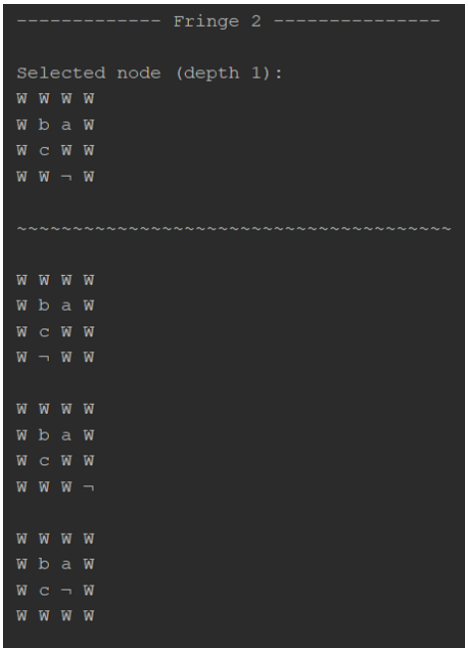
```
------------- Fringe 2749803 ---------------

Selected node (depth 13):
W W W W
W W W W
W W W ¬
a b c W

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

W W W W
W W W W
W W ¬ W
a b c W

W W W W
W W W ¬
W W W W
a b c W

W W W W
W W W W
W W W W
a b c ¬

------------- Fringe 2749804 ---------------

Exception in thread "main" java.lang.OutOfMemoryError: Java heap space <1 internal call>
    at java.base/java.lang.invoke.StringConcatFactory$MethodHandleInlineCopyStrategy.newArray(StringConcatFactory.java:1633)
    at java.base/java.lang.invoke.DirectMethodHandle$Holder.invokeStatic(DirectMethodHandle$Holder)
    at java.base/java.lang.invoke.LambdaForm$MH/0x00000008000d0440.invoke(LambdaForm$MH)
    at java.base/java.lang.invoke.LambdaForm$MH/0x00000008000c1840.linkToTargetMethod(LambdaForm$MH)
    at Tree.Grid.toString(Grid.java:230)
    at java.base/java.lang.String.valueOf(String.java:2951)
    at java.base/java.lang.invoke.DirectMethodHandle$Holder.invokeStatic(DirectMethodHandle$Holder)
    at java.base/java.lang.invoke.LambdaForm$MH/0x00000008000c1c40.invoke(LambdaForm$MH)
    at java.base/java.lang.invoke.LambdaForm$MH/0x00000008000db440.invoke(LambdaForm$MH)
    at java.base/java.lang.invoke.Invokers$Holder.linkToTargetMethod(Invokers$Holder)
    at Search.Search.expandNode(Search.java:48)
    at Search.BFS.search(BFS.java:31)
    at Main.main(Main.java:96)
```

2.2 Depth First Search (DFS)

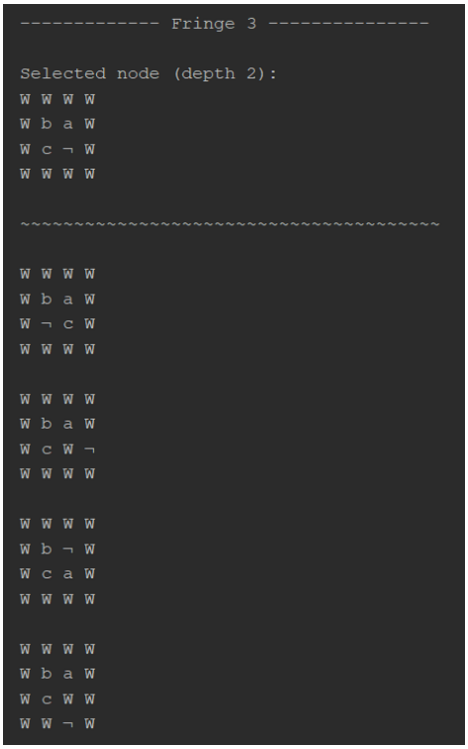
Example 1 (depth 5):

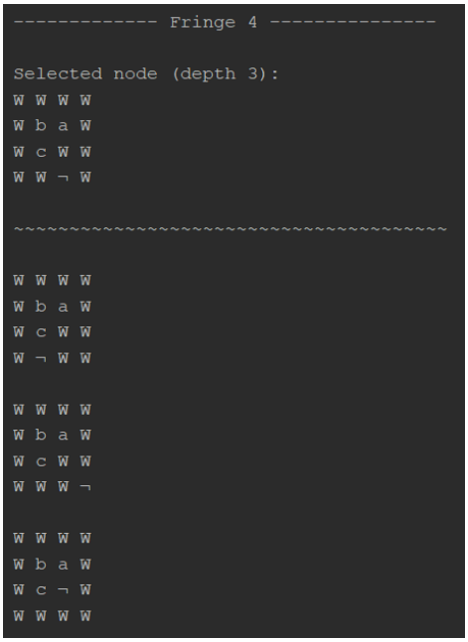
Implementing DFS for this type of problem has an obvious flaw in that the nodes are generated in an ordered manner. In the case of my algorithm, this is left, right, up and down. As such, the node added to the top of the **Stack** will always be the last possible direction. In this particular example, the agent will keep moving down and then back up again, since these will be the latest nodes generated.



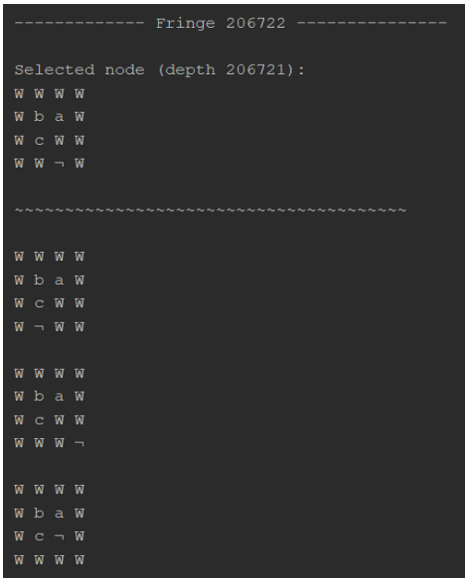


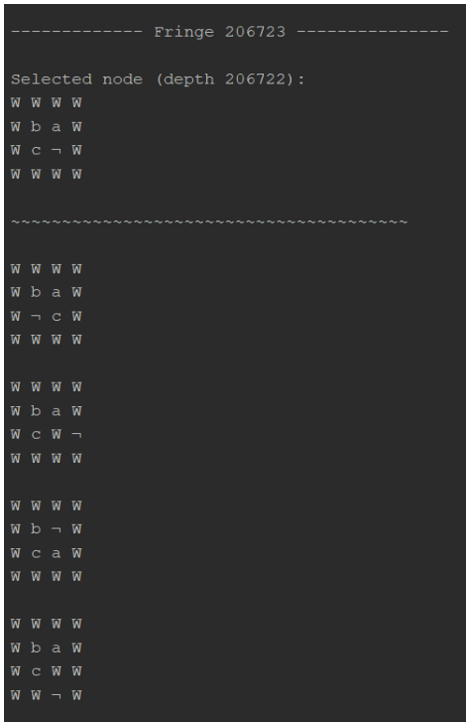
This means that unless modified accordingly, i.e. so the order of the generated nodes is randomised, the algorithm will almost certainly never find a solution.





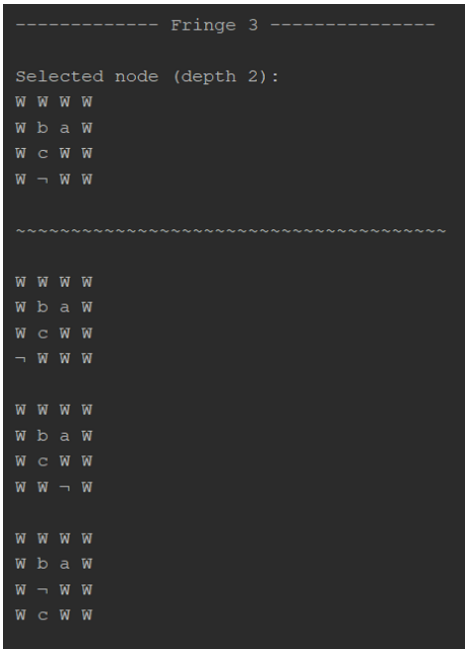
Eventually, this will cause an **OutOfMemoryException**, since it will repeat this process indefinitely, with the only difference being that the depth has increased by one each time.

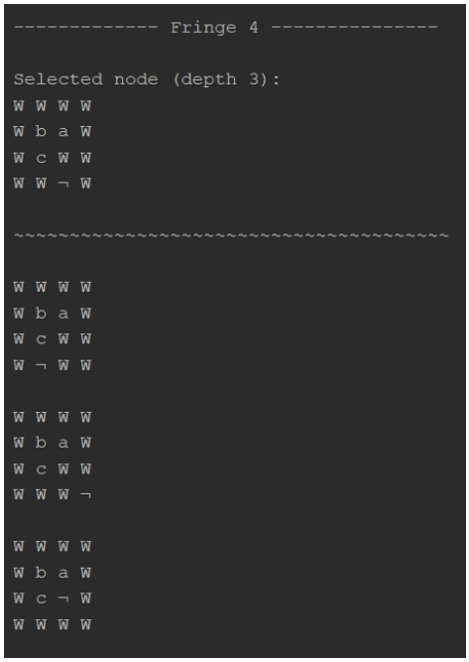




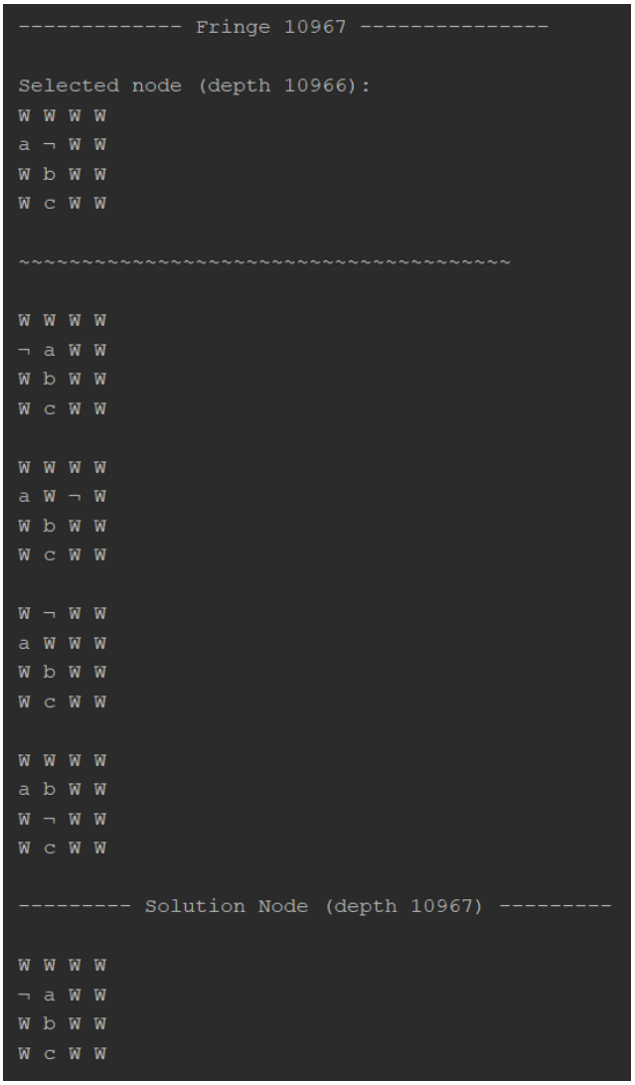
Example 2 (depth 5 with random order):

In order to get DFS working with this problem, the order in which the generated nodes are placed onto the **Stack** should be randomised. This is done by using the **Collections.shuffle()** function on the children list. Below, we can see the effect this has on the third and fourth node expansion.





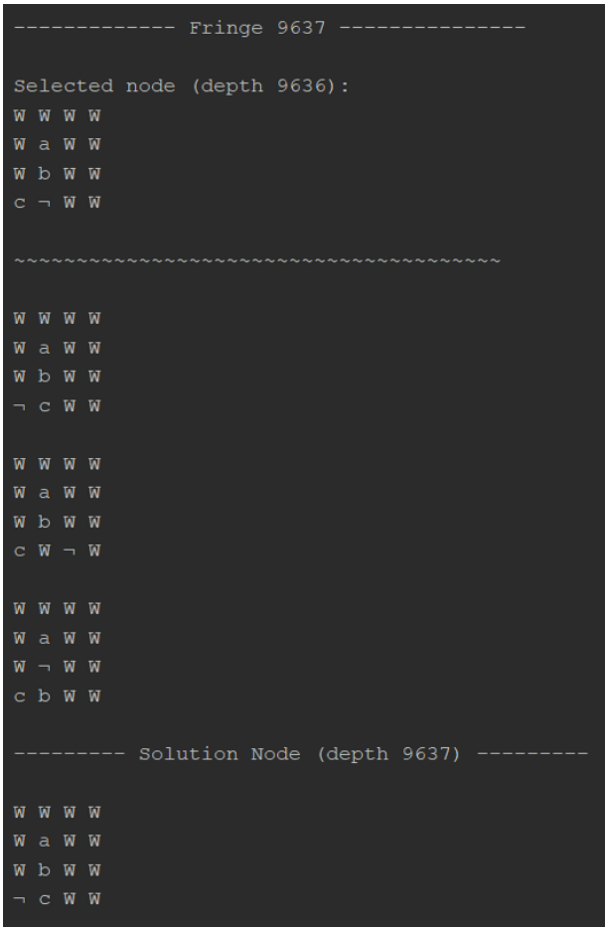
By completely arbitrary movements, the algorithm will eventually find some solution.



However, this solution is almost guaranteed to not be the optimal solution. The solution returned here is in **10967** steps, whereas the optimal solution for this problem is in merely 5.

```
Solution found after generating 34643 nodes: UP -> RIGHT -> LEFT -> UP -> DOWN -> DOWN -> DOWN -> LEFT -> LEFT -> RIGHT -> RIGHT -> LEFT -> UP -> RIGHT -> RIGHT ->
Depth of solution found: 10967
```

Due to the random nature of this algorithm, the solution returned is very likely to be different each time, with the below showing what the algorithm returns on another run-through.



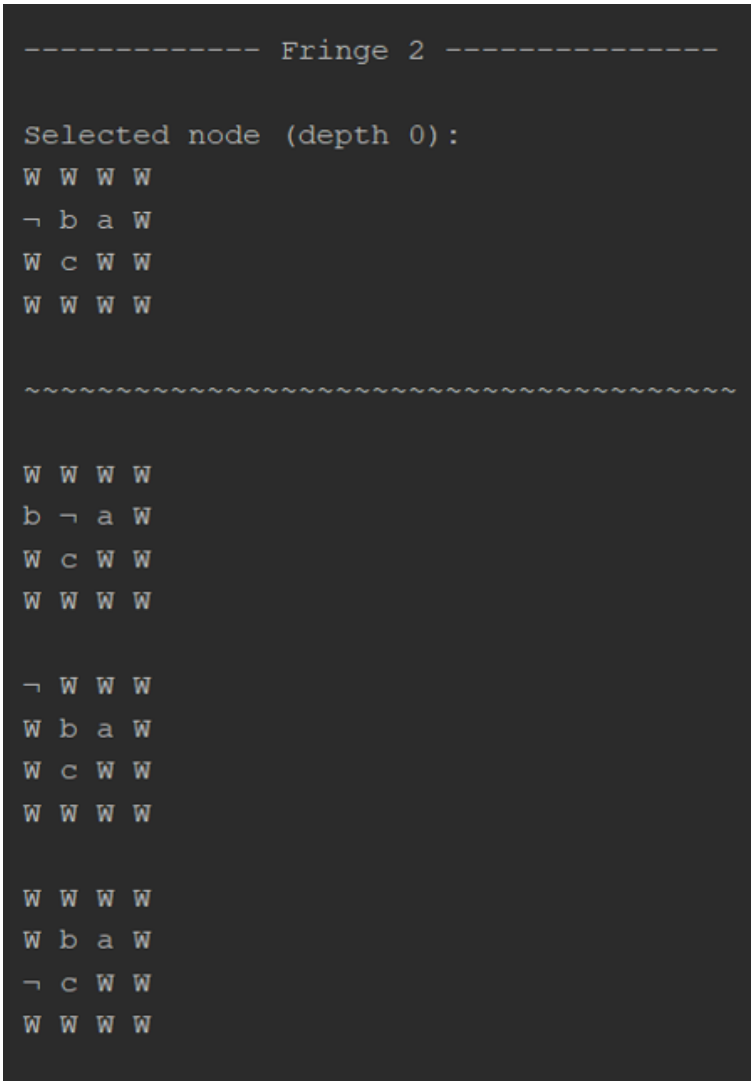
2.3 Iterative Deepening Search (IDS)

Example 1 (depth 6):

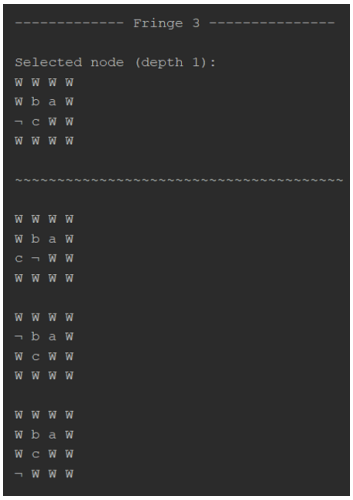
The IDS algorithm begins by calling DLS with a limit of depth 1. The agent in this initial state has three possible directions that it can move in. Each of these children are then also checked to see if they are the goal state.

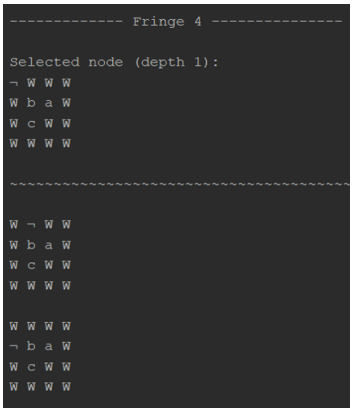
```
*********************************************************************************************
Iterative Deepening Search (IDS) - Optimal Solution Depth: 6
*********************************************************************************************

Root:
W W W W
¬ b a W
W c W W
W W W W

------------- Fringe 1 ---------------

Selected node (depth 0):
W W W W
¬ b a W
W c W W
W W W W

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

W W W W
b ¬ a W
W c W W
W W W W

¬ W W W
W b a W
W c W W
W W W W

W W W W
W b a W
¬ c W W
W W W W
```

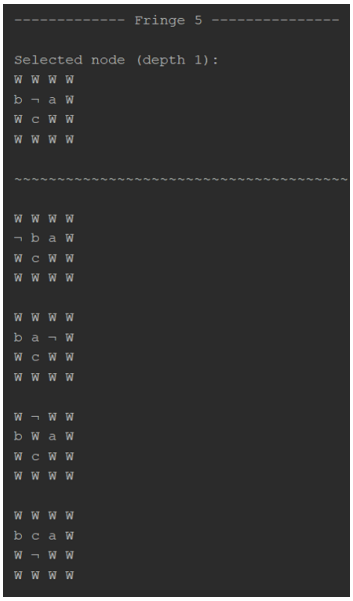
The IDS then increases the limit to 2, and performs another DLS on this limit.



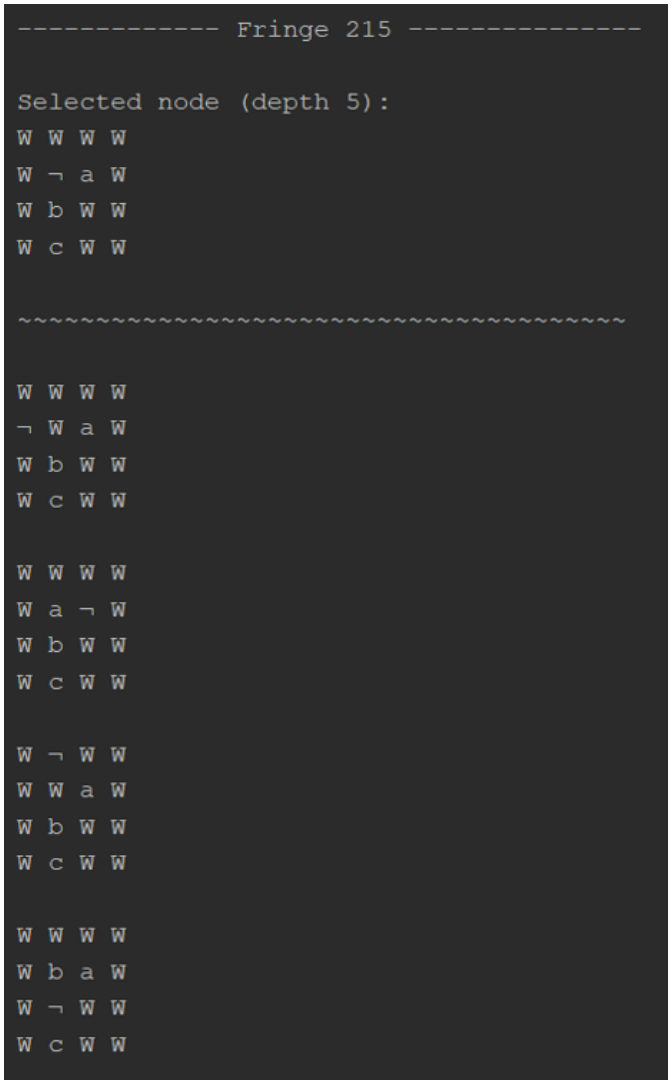
However, because the limit is now 2, we also expand the nodes at depth 1, and then check the nodes at depth 2 that are generated from these expansions.



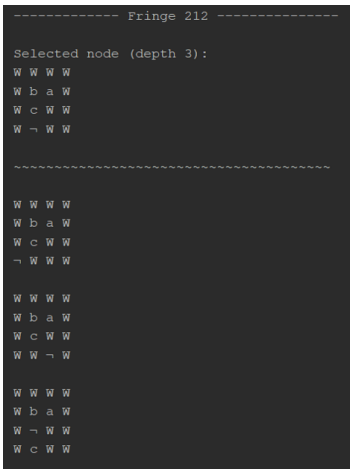


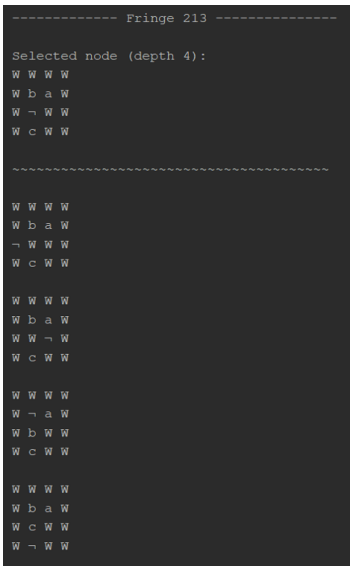


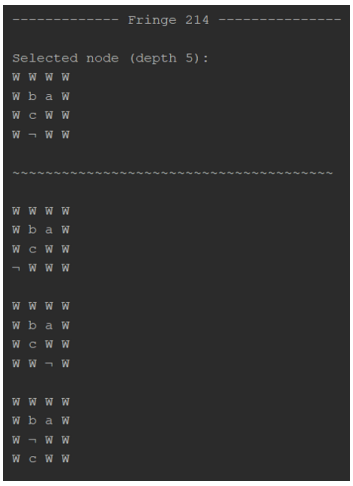
Since none of these nodes contain the goal state (which can actually be found at depth 6), we repeat this process until the goal state is found.



If we look at the node expansions prior to this, we can see the DFS nature of the algorithm, as the next node selected is always the last generated. The next node expansion contains the goal state that was seen in the previous image (labelled **Fringe 215**).







The optimal solution was therefore found at depth 6.

```
Depth 1:
W W W W
W b a W
¬ c W W
W W W W

Depth 2:
W W W W
W b a W
W c W W
¬ W W W

Depth 3:
W W W W
W b a W
W c W W
W ¬ W W

Depth 4:
W W W W
W b a W
W ¬ W W
W c W W

Depth 5:
W W W W
W ¬ a W
W b W W
W c W W

Depth 6:
W W W W
W a ¬ W
W b W W
W c W W

Solution found after generating 689 nodes: DOWN -> DOWN -> RIGHT -> UP -> UP -> RIGHT
Depth of solution found: 6
```

Example 2 (depth 15):

Whilst IDS has a better space complexity than BFS because it works in a DFS manner with its node expansion, it is not immune to space constraints at a large enough problem size. The following problem has an optimal solution at a depth of 15.

```
********************************************************************************
Iterative Deepening Search (IDS) - Optimal Solution Depth: 15
********************************************************************************

Root:
¬ W W W
c b a W
W W W W
W W W W

------------ Fringe 1 ---------------

Selected node (depth 0):
¬ W W W
c b a W
W W W W
W W W W

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

W ¬ W W
c b a W
W W W W
W W W W

c W W W
¬ b a W
W W W W
W W W W
```

This means that like BFS, a failed IDS at a certain depth will cause a **Java heap space error**.

```
------------ Fringe 4294842 ---------------

Selected node (depth 13):
W W W ¬
c b a W
W W W W
W W W W

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

W W ¬ W
c b a W
W W W W
W W W W

Exception in thread "main" java.lang.OutOfMemoryError: Java heap space <1 internal call>
    at java.base/java.lang.invoke.StringConcatFactory$MethodHandleInlineCopyStrategy.newArray(StringConcatFactory.java:1633)
    at java.base/java.lang.invoke.DirectMethodHandle$Holder.invokeStatic(DirectMethodHandle$Holder)
    at java.base/java.lang.invoke.LambdaForm$MH/0x00000008000d0440.invoke(LambdaForm$MH)
    at java.base/java.lang.invoke.LambdaForm$MH/0x00000008000c7040.linkToTargetMethod(LambdaForm$MH)
    at Tree.Grid.toString(Grid.java:230)
    at java.base/java.lang.String.valueOf(String.java:2951)
    at java.base/java.io.PrintStream.println(PrintStream.java:897)
    at Search.Search.expandNode(Search.java:76)
    at Search.IDS.depthLimitedSearch(IDS.java:51)
    at Search.IDS.search(IDS.java:26)
    at Main.main(Main.java:96)
```

2.4 A* Heuristic Search

Example 1 (depth 3):

A* Search works differently to the other searches in the sense that it is an informed search meaning the algorithm has additional information in regards to what node it should select next to expand.

Looking through a simple example, we can see that a solution that has a depth of 3 can be found by expanding 3 nodes.

```
********************************************************************************
A* Heuristic Search - Optimal Solution Depth: 3
********************************************************************************

Root:
W W W W
W b a W
W c W W
W ¬ W W

------------ Fringe 1 ---------------

Selected node (depth 0):
W W W W
W b a W
W c W W
W ¬ W W

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

W W W W
W b a W
W c W W
¬ W W W

W W W W
W b a W
W c W W
W W ¬ W

W W W W
W b a W
W ¬ W W
W c W W

===== A* Heuristic Search information =====

Depth at 1

Child 1:
Heuristic (Manhattan distance): 3
Evaluation function value: 4

Child 2:
Heuristic (Manhattan distance): 3
Evaluation function value: 4

Child 3:
Heuristic (Manhattan distance): 2
Evaluation function value: 3
```

Since the third child has the lowest evaluation function value (current depth of 1 plus the heuristic of 2), the **PriorityQueue** prioritises that as the next node, and thus it is selected as the next node to be expanded.

```
------------- Fringe 2 ---------------

Selected node (depth 1):
W W W W
W b a W
W ¬ W W
W c W W

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

W W W W
W b a W
¬ W W W
W c W W

W W W W
W b a W
W W ¬ W
W c W W

W W W W
W ¬ a W
W b W W
W c W W

W W W W
W b a W
W c W W
W ¬ W W

===== A* Heuristic Search information =====

Depth at 2

Child 1:
Heuristic (Manhattan distance): 2
Evaluation function value: 4

Child 2:
Heuristic (Manhattan distance): 2
Evaluation function value: 4

Child 3:
Heuristic (Manhattan distance): 1
Evaluation function value: 3

Child 4:
Heuristic (Manhattan distance): 3
Evaluation function value: 5
```

In the next step, we select the third child from the **Fringe 2** node expansion seen above. Despite the fact that it has a higher depth than some of the other possible nodes that can be expanded, its evaluation function is lower thanks to the fact that the heuristic is merely 1.

```
------------- Fringe 3 ---------------

Selected node (depth 2):
W W W W
W ¬ a W
W b W W
W c W W

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

W W W W
¬ W a W
W b W W
W c W W

W W W W
W a ¬ W
W b W W
W c W W

W ¬ W W
W W a W
W b W W
W c W W

W W W W
W b a W
W ¬ W W
W c W W

===== A* Heuristic Search information =====

Depth at 3

Child 1:
Heuristic (Manhattan distance): 1
Evaluation function value: 4

Child 2:
Heuristic (Manhattan distance): 0
Evaluation function value: 3

Child 3:
Heuristic (Manhattan distance): 1
Evaluation function value: 4

Child 4:
Heuristic (Manhattan distance): 2
Evaluation function value: 5
```

We have now found a solution, which the **PriorityQueue** will select regardless as it has the lowest evaluation function value.

```
--------- Solution Node (depth 3) ---------

W W W W
W a ¬ W
W b W W
W c W W

-----------------------------------------

Depth 1:
W W W W
W b a W
W ¬ W W
W c W W

Depth 2:
W W W W
W ¬ a W
W b W W
W c W W

Depth 3:
W W W W
W a ¬ W
W b W W
W c W W

Solution found after generating 12 nodes: UP -> UP -> RIGHT
Depth of solution found: 3
```

Example 2 (depth 15):

Since A* is a smarter searching algorithm, it is capable of solving problems that the other uninformed search methods cannot due to **Java heap space errors**. For instance, it is able to find a solution for the problem that has an optimal solution at depth 15 that IDS was unable to find.

```
********************************************************************************
A* Heuristic Search - Optimal Solution Depth: 15
********************************************************************************

Root:
¬ W W W
c b a W
W W W W
W W W W

------------- Fringe 1 ---------------

Selected node (depth 0):
¬ W W W
c b a W
W W W W
W W W W

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

W ¬ W W
c b a W
W W W W
W W W W

c W W W
¬ b a W
W W W W
W W W W

===== A* Heuristic Search information =====

Depth at 1

Child 1:
Heuristic (Manhattan distance): 5
Evaluation function value: 6

Child 2:
Heuristic (Manhattan distance): 6
Evaluation function value: 7
```

The next node that is expanded is the one with the lowest value from the evaluation function, which was the first child in this case.

```
------------- Fringe 2 ---------------

Selected node (depth 1):
W ¬ W W
c b a W
W W W W
W W W W

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

¬ W W W
c b a W
W W W W
W W W W

W W ¬ W
c b a W
W W W W
W W W W

W b W W
c ¬ a W
W W W W
W W W W

===== A* Heuristic Search information =====

Depth at 2

Child 1:
Heuristic (Manhattan distance): 5
Evaluation function value: 7

Child 2:
Heuristic (Manhattan distance): 5
Evaluation function value: 7

Child 3:
Heuristic (Manhattan distance): 6
Evaluation function value: 8
```

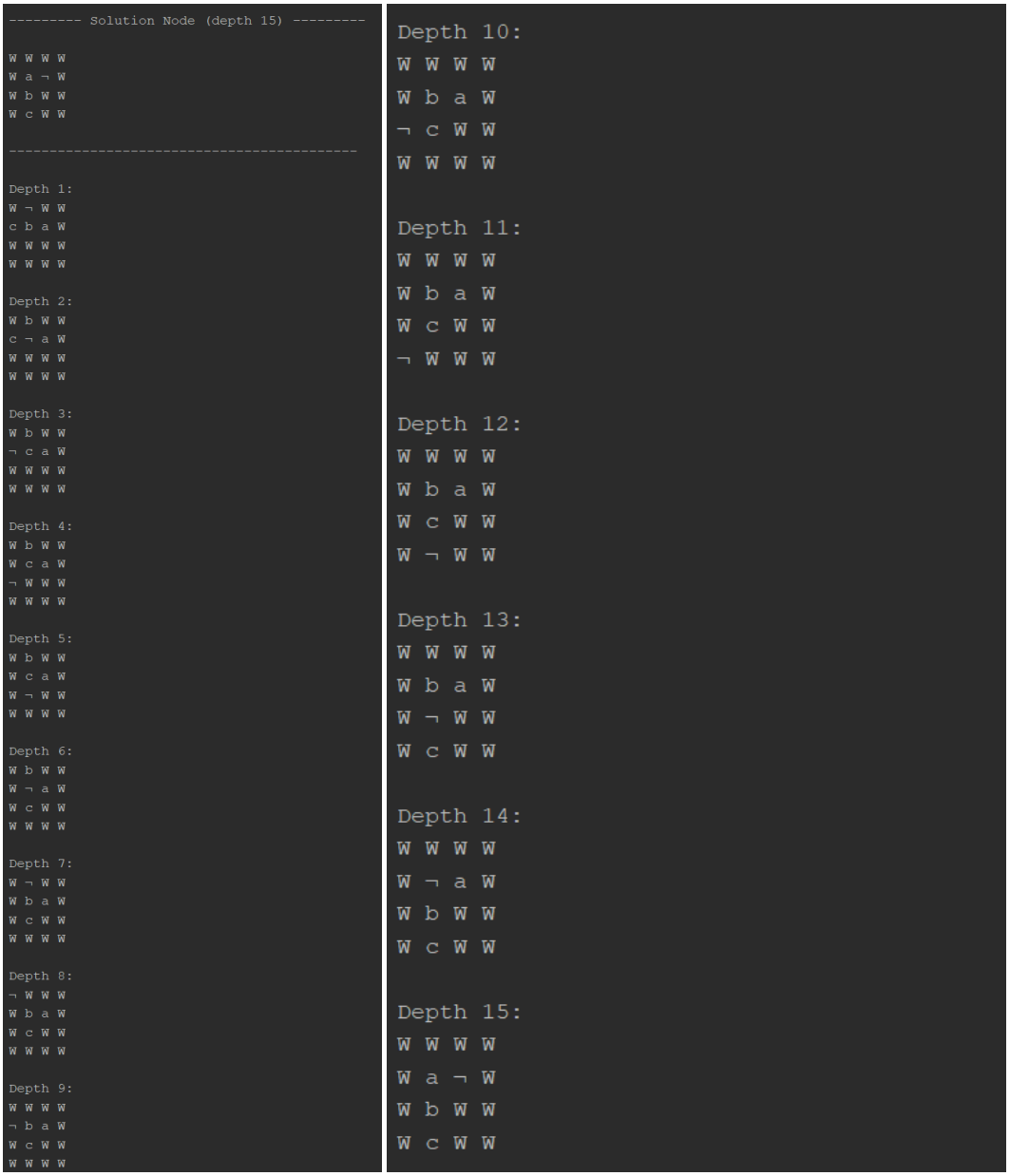
This process continues until a solution is found - here is the last node expansion performed before the goal state is discovered.

```
------------ Fringe 32387 --------------

Selected node (depth 10):
W b W W
c a W W
W W ¬ W
W W W W

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

W b W W
c a W W
W ¬ W W
W W W W

W b W W
c a W W
W W W ¬
W W W W

W b W W
c a ¬ W
W W W W
W W W W

W b W W
c a W W
W W W W
W W ¬ W

===== A* Heuristic Search information =====

Depth at 11

Child 1:
Heuristic (Manhattan distance): 5
Evaluation function value: 16

Child 2:
Heuristic (Manhattan distance): 5
Evaluation function value: 16

Child 3:
Heuristic (Manhattan distance): 5
Evaluation function value: 16

Child 4:
Heuristic (Manhattan distance): 5
Evaluation function value: 16
```

You would have noticed that the last expanded node does not contain the solution - this is because these nodes have an equal evaluation function value, so the way that they are selected in this situation is completely down to the implementation of the Java **PriorityQueue** structure. The solution node can actually be seen being generated in the node generation labelled **Fringe 32385**, two nodes before the point at which the final expansion took place.

```
------------- Fringe 32385 ---------------

Selected node (depth 14):
W W W W
W ¬ a W
W b W W
W c W W

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

W W W W
¬ W a W
W b W W
W c W W

W W W W
W a ¬ W
W b W W
W c W W

W ¬ W W
W W a W
W b W W
W c W W

W W W W
W b a W
W ¬ W W
W c W W

===== A* Heuristic Search information =====

Depth at 15

Child 1:
Heuristic (Manhattan distance): 1
Evaluation function value: 16

Child 2:
Heuristic (Manhattan distance): 0
Evaluation function value: 15

Child 3:
Heuristic (Manhattan distance): 1
Evaluation function value: 16

Child 4:
Heuristic (Manhattan distance): 2
Evaluation function value: 17
```

The following solution is then found, which corresponds to the optimal solution that exists at 15 steps.



```
--------- Solution Node (depth 15) ---------

W W W W
W a ¬ W
W b W W
W c W W

----------------------------------------

Depth 1:
W ¬ W W
c b a W
W W W W
W W W W

Depth 2:
W b W W
c ¬ a W
W W W W
W W W W

Depth 3:
W b W W
¬ c a W
W W W W
W W W W

Depth 4:
W b W W
W c a W
¬ W W W
W W W W

Depth 5:
W b W W
W c a W
W ¬ W W
W W W W

Depth 6:
W b W W
W ¬ a W
W c W W
W W W W

Depth 7:
W ¬ W W
W b a W
W c W W
W W W W

Depth 8:
¬ W W W
W b a W
W c W W
W W W W

Depth 9:
W W W W
¬ b a W
W c W W
W W W W

Depth 10:
W W W W
W b a W
¬ c W W
W W W W

Depth 11:
W W W W
W b a W
W c W W
¬ W W W

Depth 12:
W W W W
W b a W
W c W W
W ¬ W W

Depth 13:
W W W W
W b a W
W ¬ W W
W c W W

Depth 14:
W W W W
W ¬ a W
W b W W
W c W W

Depth 15:
W W W W
W a ¬ W
W b W W
W c W W
```

```
Solution found after generating 101800 nodes: RIGHT -> DOWN -> LEFT -> DOWN -> RIGHT -> UP -> UP -> LEFT -> DOWN -> DOWN -> DOWN -> RIGHT -> UP -> UP -> RIGHT
Depth of solution found: 15
```

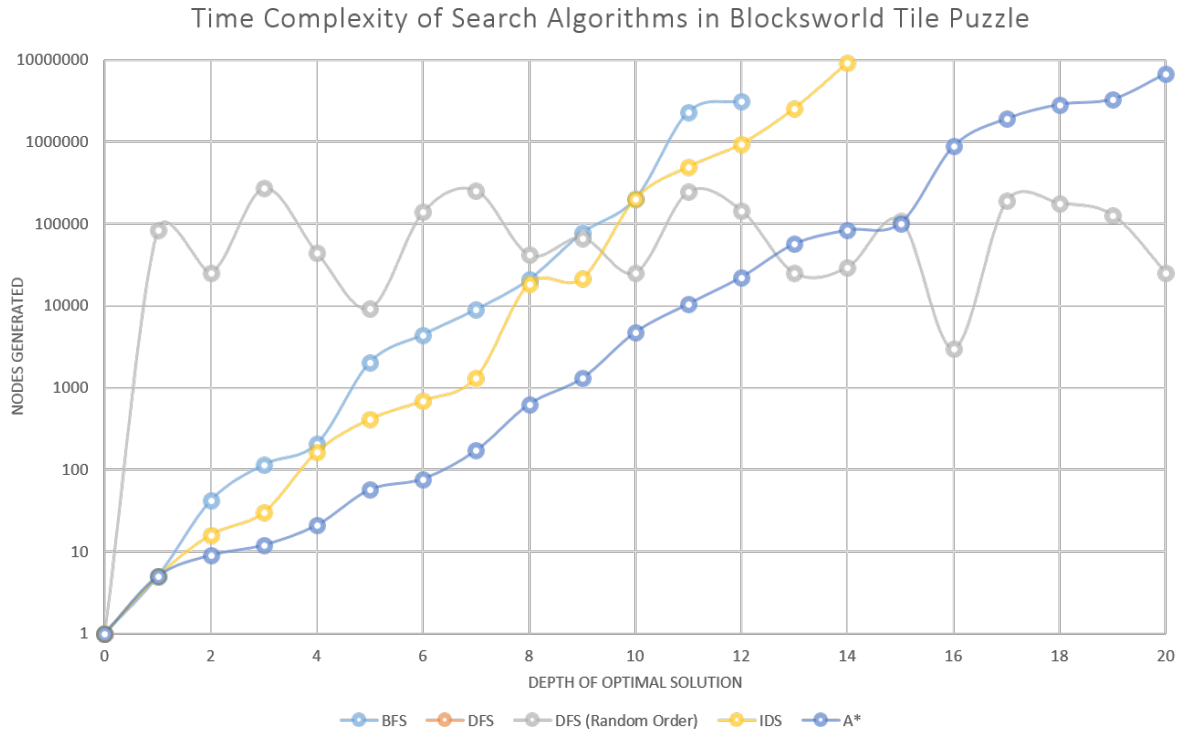
3 Scalability Study

As explained above, the scalability study was conducted using 21 states, each of which corresponded to a problem that had an optimal solution from 0 to 20. For completeness, I have attached these initial states, as seen in appendix A.

Depth of optimal solution	BFS		DFS		DFS (Random Order)	
	Nodes generated	Depth	Nodes generated	Depth	Nodes generated	Depth
0	1	0	1	0	1	1
1	5	1	Heap space error	N/A	84863	26798
2	42	2	Heap space error	N/A	25274	8012
3	115	3	Heap space error	N/A	270143	85111
4	208	4	Heap space error	N/A	44338	14000
5	2047	5	Heap space error	N/A	9288	2923
6	4445	6	Heap space error	N/A	138676	43652
7	9004	7	Heap space error	N/A	254565	80381
8	20932	8	Heap space error	N/A	41730	13157
9	77169	9	Heap space error	N/A	67296	21240
10	202095	10	Heap space error	N/A	25297	7963
11	2318900	11	Heap space error	N/A	245155	77368
12	3127458	12	Heap space error	N/A	144702	45603
13	Heap space error	N/A	Heap space error	N/A	24998	7909
14	Heap space error	N/A	Heap space error	N/A	29386	9258
15	Heap space error	N/A	Heap space error	N/A	109742	34647
16	Heap space error	N/A	Heap space error	N/A	3001	951
17	Heap space error	N/A	Heap space error	N/A	190469	60053
18	Heap space error	N/A	Heap space error	N/A	178706	56441
19	Heap space error	N/A	Heap space error	N/A	127710	40242
20	Heap space error	N/A	Heap space error	N/A	25389	8040

Depth of optimal solution	IDS		A*	
	Nodes generated	Depth	Nodes generated	Depth
0	1	0	1	0
1	5	1	5	1
2	16	2	9	2
3	30	3	12	3
4	164	4	21	4
5	412	5	57	5
6	689	6	77	6
7	1328	7	171	7
8	18584	8	627	8
9	21386	9	1302	9
10	202171	10	4730	10
11	494670	11	10500	11
12	942293	12	22262	12
13	2559398	13	56516	13
14	9238235	14	83428	14
15	Heap space error	N/A	101800	15
16	Heap space error	N/A	883397	16
17	Heap space error	N/A	1909423	17
18	Heap space error	N/A	2844540	18
19	Heap space error	N/A	3311656	19
20	Heap space error	N/A	6777725	20

The below graph visualises these results. Take note of the logarithmic scale used for the nodes generated (y-axis).



From this data, you may extrapolate that the DFS with a random order is the best performing algorithm for problems with larger depth solutions. However, although it may be quicker to find these solutions, they are almost guaranteed to not be **optimal**. It is also worth noting that due to the arbitrary nature of this algorithm, it is actually the worst performing search at the earlier depth problems.

Looking at the other algorithms, it is clear that A* is the best performing algorithm in terms of time complexity, however it is also clear that it is exponential based on the problem size. BFS and IDS would continue to rise exponentially too if they did not run into space complexity issues, but have a worse time complexity than A*. For the most part, IDS is better than BFS, but this is dependent on where in the tree structure the goal state is located.

4 Extras and Limitations

For my extras, I implemented an algorithm (as part of the **generateStartState()** function in the **Grid** class) that places a number of collision blocks (represented by a **!**) on the grid at random spots. This excludes the start state positions of the blocks and the agent, as well as the intended position of the blocks at the end state. The exact number of collision blocks is controlled through the constant **COLLISIONS_NUM** in the **Grid** class.

The position of these blocks can potentially have a negative effect in regards to optimal depth, and a positive effect in terms of the time complexity (assuming the problem is solvable), as can be seen when performing A* Search on the default state.

```
********************************************************************************
A* Heuristic Search - Optimal Solution Depth: 14
********************************************************************************

Root:
W W W W
W W W W
W W ! W
a b c ¬
```

```
Depth 15:
W W W W
W a W W
W b ! W
W c ¬ W

Solution found after generating 22056 nodes: LEFT -> LEFT -> LEFT -> UP -> RIGHT -> DOWN -> RIGHT -> RIGHT -> UP -> UP -> LEFT -> LEFT -> DOWN -> DOWN -> RIGHT
Depth of solution found: 15
```

The solution found was actually found in **22056** nodes, as opposed to the **83248** nodes that the unmodified version of this problem had for A* Search.

I also implemented an alternative heuristic for the A* Search algorithm in the form of the **Euclidean distance** algorithm. This is switched on by means of constant in the **Grid** class - **EUCLID**. The **calculateEuclideanDistance()** function is then used as opposed to the standard **calculateManhattanDistance()** function.

```
********************************************************************************
A* Heuristic Search - Optimal Solution Depth: 4
********************************************************************************

Root:
W W W W
W b a W
W c W W
W W ¬ W
```

The heuristic is clearly different at the third node expansion, as the **Manhattan distance** is always an integer.

```
------------ Fringe 3 --------------

Selected node (depth 1):
W W W W
W b a W
W c ¬ W
W W W W

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

W W W W
W b a W
W ¬ c W
W W W W

W W W W
W b a W
W c W ¬
W W W W

W W W W
W b ¬ W
W c a W
W W W W

W W W W
W b a W
W c W W
W W ¬ W
```

```
===== A* Heuristic Search information =====

Depth at 2

Child 1:
Heuristic: 3.414213562373095
Evaluation function value: 5.414213562373095

Child 2:
Heuristic: 3.0
Evaluation function value: 5.0

Child 3:
Heuristic: 3.414213562373095
Evaluation function value: 5.414213562373095

Child 4:
Heuristic: 3.0
Evaluation function value: 5.0
```

The solution that it finds in this case is identical to what was found with the **Manhattan distance** heuristic, however other problems may perform better or worse if this heuristic is used.

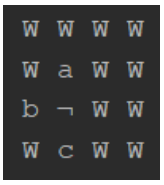
```
--------- Solution Node (depth 4) ---------

W W W W
W a ¬ W
W b W W
W c W W

-----------------------------------------

Depth 1:
W W W W
W b a W
W c W W
W ¬ W W

Depth 2:
W W W W
W b a W
W ¬ W W
W c W W

Depth 3:
W W W W
W ¬ a W
W b W W
W c W W

Depth 4:
W W W W
W a ¬ W
W b W W
W c W W

Solution found after generating 21 nodes: LEFT -> UP -> UP -> RIGHT
Depth of solution found: 4
```

In regards to limitations, the space complexity of my solution could be significantly improved if I were to refactor the **Grid** class to use something other than a **HashMap** to store the **blockPositions**. In terms of the scalability study, I could have ran the DFS algorithm on each of the states multiple times so that a more accurate average could be taken. I also could have tested A* and DFS at higher depths to determine at what point each of them would fail to find a solution due to space constraints.

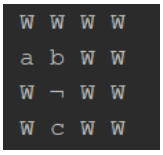
Appendices

A Problem Start States

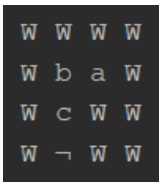
State 0



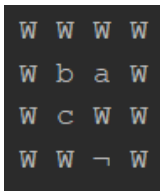
State 1



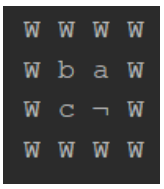
State 2



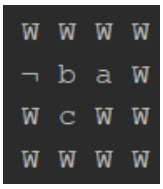
State 3



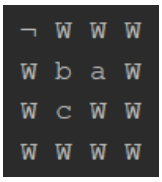
State 4



State 5

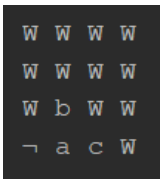


State 6

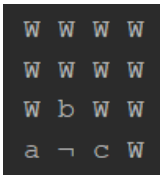


State 7

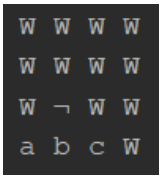
State 8



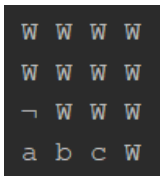
State 9



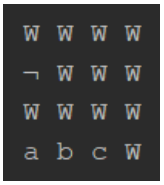
State 10



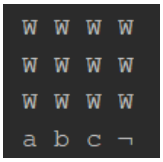
State 11



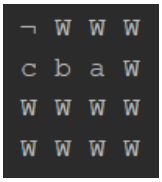
State 12



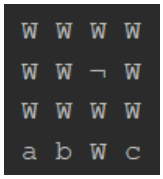
State 13



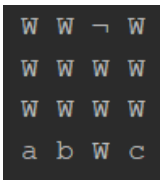
State 14



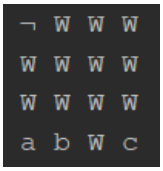
State 15



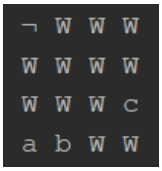
State 16



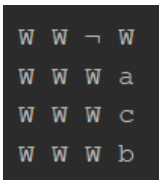
State 17



State 18



State 19



State 20

B Code

B.1 Node

```
package Tree;

import java.util.ArrayList;
import java.util.List;

// Class that defines a Node, which is used to form a Tree structure by means of
    ↪ the parent and children variables.
public class Node implements Comparable<Node>
{
	private final Grid value;
	private final Node parent;
	private final List<Node> children;
	private double estimatedCost;

	public Node(Grid value, Node parent)
	{
		this.value = value;
		this.parent = parent;
		this.children = new ArrayList<>();
	}

	public Grid getValue()
	{
		return value;
	}

	public Node getParent()
	{
		return parent;
	}

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

	// Method that returns the depth of the Node in the tree.
	public int getDepth()
	{
		int depth = 0;
		Node node = this;

	while (node.parent != null)
	{
```

```
                depth++;
                node = node.parent;
        }

                return depth;
        }

        public void setEstimatedCost(double estimatedCost)
        {
                this.estimatedCost = estimatedCost;
        }

        public double getEstimatedCost()
        {
                return estimatedCost;
        }

        public void addChild(Node child)
        {
                this.children.add(child);
        }

        // compareTo() method that defines how Nodes work as a Comparable, which is
            ↪  necessary for use in a PriorityQueue.
        public int compareTo(Node node)
        {
                return Double.compare(this.getEstimatedCost(), node.getEstimatedCost
                    ↪ ());
        }
}
```

B.2 Grid

```
package Tree;

import java.util.*;

// Class that stores the information about the grid and its current state, and the
    ↪ methods that can be used to manipulate it.
public class Grid
{
	private final int size;
	private final char[][] grid;
	private int agentPosition;
	private Direction lastDirection = null;
	private Map<Character, Integer> blockPositions = new HashMap<>();

	private static final char A = '¬';
	private static final char E = 'W';
	private static final char C = '!';

	private static final int COLLISIONS_NUM = 0;
	private static final boolean EUCLID = false;

	public Grid(int size)
	{
		this.size = size;
		this.grid = new char[size][size];

		for (int i = 0; i < size; i++)
		{
			for (int j = 0; j < size; j++)
			{
				grid[i][j] = E;
			}
		}
	}

	public Grid(char[][] grid, int agentPosition, Map<Character, Integer>
	    ↪ blockPositions)
	{
		this.size = grid.length;
		this.grid = grid;
		this.agentPosition = agentPosition;
		this.blockPositions = blockPositions;
	}

	// Method that automatically generates a starting grid based on its size.
	public void generateStartingGrid()
```

```
{
        int blockNumber = size - 1;
        char block = 'a';

        blockPositions.clear();

        for (int i = 0; i < blockNumber; i++)
        {
                setBlockPosition(block, i, size - 1);
                block++;
        }

        setAgentPosition(size - 1, size - 1);

        // If collisions are turned on.
        if (COLLISIONS_NUM > 0)
        {
                List<Integer> possiblePositions = new ArrayList<>();

                // Collect a list of all possible locations - i.e. any
                    ↪ location that is not needed for the agent and blocks
                    ↪ in the start state, or the blocks in the solution
                    ↪ state.
                for (int i = 0; i < (size * (size - 1)); i++)
                {
                        if ((i - 1) % size != 0 || i == 1)
                                possiblePositions.add(i);
                }

                Random random = new Random();

                // Randomly select an index of the possible positions, add
                    ↪ the collision block, and then remove it from the index
                    ↪ .
                for (int i = 0; i < COLLISIONS_NUM; i++)
                {
                        int index = random.nextInt(possiblePositions.size());
                        int position = possiblePositions.get(index);
                        addCollisionBlock(getXCoord(position), getYCoord(
                            ↪ position));
                        possiblePositions.remove(index);
                }
        }
}

// IMPORTANT: This function only works if the size of the grid is 4x4, i.e.
    ↪  size = 4.
```

```
// Method that sets a custom starting position based on the depth value
    ↪ provided.
public void generateStartingGrid(int depth)
{
        blockPositions.clear();

        switch (depth)
        {
                case 0:
                        setBlockPosition('a', 1, 1);
                        setBlockPosition('b', 1, 2);
                        setBlockPosition('c', 1, 3);
                        setAgentPosition(2, 1);
                        break;
                case 1:
                        setBlockPosition('a', 1, 1);
                        setBlockPosition('b', 0, 2);
                        setBlockPosition('c', 1, 3);
                        setAgentPosition(1, 2);
                        break;
                case 2:
                        setBlockPosition('a', 0, 1);
                        setBlockPosition('b', 1, 1);
                        setBlockPosition('c', 1, 3);
                        setAgentPosition(1, 2);
                        break;
                case 3:
                        setBlockPosition('a', 2, 1);
                        setBlockPosition('b', 1, 1);
                        setBlockPosition('c', 1, 2);
                        setAgentPosition(1, 3);
                        break;
                case 4:
                        setBlockPosition('a', 2, 1);
                        setBlockPosition('b', 1, 1);
                        setBlockPosition('c', 1, 2);
                        setAgentPosition(2, 3);
                        break;
                case 5:
                        setBlockPosition('a', 2, 1);
                        setBlockPosition('b', 1, 1);
                        setBlockPosition('c', 1, 2);
                        setAgentPosition(2, 2);
                        break;
                case 6:
                        setBlockPosition('a', 2, 1);
                        setBlockPosition('b', 1, 1);
                        setBlockPosition('c', 1, 2);
```

```
            setAgentPosition(0, 1);
            break;
        case 7:
            setBlockPosition('a', 2, 1);
            setBlockPosition('b', 1, 1);
            setBlockPosition('c', 1, 2);
            setAgentPosition(0, 0);
            break;
        case 8:
            setBlockPosition('a', 2, 1);
            setBlockPosition('b', 1, 1);
            setBlockPosition('c', 1, 2);
            setAgentPosition(1, 0);
            break;
        case 9:
            setBlockPosition('a', 1, 3);
            setBlockPosition('b', 1, 2);
            setBlockPosition('c', 2, 3);
            setAgentPosition(0, 3);
            break;
        case 10:
            setBlockPosition('a', 0, 3);
            setBlockPosition('b', 1, 2);
            setBlockPosition('c', 2, 3);
            setAgentPosition(1, 3);
            break;
        case 11:
            setBlockPosition('a', 0, 3);
            setBlockPosition('b', 1, 3);
            setBlockPosition('c', 2, 3);
            setAgentPosition(1, 2);
            break;
        case 12:
            setBlockPosition('a', 0, 3);
            setBlockPosition('b', 1, 3);
            setBlockPosition('c', 2, 3);
            setAgentPosition(0, 2);
            break;
        case 13:
            setBlockPosition('a', 0, 3);
            setBlockPosition('b', 1, 3);
            setBlockPosition('c', 2, 3);
            setAgentPosition(0, 1);
            break;
        case 14:
            generateStartingGrid();
            break;
        case 15:
```

```
                        setBlockPosition('a', 2, 1);
                        setBlockPosition('b', 1, 1);
                        setBlockPosition('c', 0, 1);
                        setAgentPosition(0, 0);
                        break;
                case 16:
                        setBlockPosition('a', 0, 3);
                        setBlockPosition('b', 1, 3);
                        setBlockPosition('c', 3, 3);
                        setAgentPosition(2, 1);
                        break;
                case 17:
                        setBlockPosition('a', 0, 3);
                        setBlockPosition('b', 1, 3);
                        setBlockPosition('c', 3, 3);
                        setAgentPosition(2, 0);
                        break;
                case 18:
                        setBlockPosition('a', 0, 3);
                        setBlockPosition('b', 1, 3);
                        setBlockPosition('c', 3, 3);
                        setAgentPosition(0, 0);
                        break;
                case 19:
                        setBlockPosition('a', 0, 3);
                        setBlockPosition('b', 1, 3);
                        setBlockPosition('c', 3, 2);
                        setAgentPosition(0, 0);
                        break;
                case 20:
                        setBlockPosition('a', 3, 1);
                        setBlockPosition('b', 3, 3);
                        setBlockPosition('c', 3, 2);
                        setAgentPosition(2, 0);
                        break;
        }
}

// Method that automatically generates the solution state.
public void generateSolutionGrid()
{
        int blockNumber = size - 1;
        char block = (char) ('a' + blockNumber - 1);

        blockPositions.clear();

        for (int i = blockNumber; i > 0; i--)
        {
```

```
                setBlockPosition(block, 1, i);
                block--;
        }
}

// Function that returns the grid.
// Since multidimensional arrays are treated as objects in Java, we need to
    ↪ make a new one and return that to avoid any conflicts between Grid
    ↪ objects.
public char[][] getGrid()
{
        char[][] newGrid = new char[size][size];

        for (int i = 0; i < size; i++)
        {
                System.arraycopy(grid[i], 0, newGrid[i], 0, grid[i].length);
        }

        return newGrid;
}

// Overridden toString() method to define how a Grid object should be
    ↪ output as a String.
@Override
public String toString()
{
        String printString = "";

        for (int i = 0; i < size; i++)
        {
                for (int j = 0; j < size; j++)
                {
                        printString = printString + grid[i][j];

                        if (j != size - 1)
                                printString = printString + "␣";
                }

                printString = printString + "\n";
        }

        return printString;
}

// Function that sets a position on the grid to a block.
private void setBlockPosition(char block, int x, int y)
{
        grid[y][x] = block;
```

```
        // If the block isn't the empty block, or the agent block, then we
            ↪ want to update where it is in the blockPositions HashMap.
        if (block != E && block != A)
        {
                blockPositions.put(block, x + (size * y));
        }
}

// Method that sets the position of the agent on the grid.
private void setAgentPosition(int x, int y)
{
        grid[y][x] = A;
        agentPosition = x + (size * y);
}

private int getXCoord(int position)
{
        return position % size;
}

private int getYCoord(int position)
{
        return position / size;
}

public int getAgentPosition()
{
        return agentPosition;
}

// Function that returns the block position of any entry in the
    ↪ blockPositions HashMap.
private int getBlockPosition(char block)
{
        if (blockPositions.containsKey(block))
                return blockPositions.get(block);

        return -1;
}

public Map<Character, Integer> getBlockPositions()
{
        return blockPositions;
}

// Method that moves the agent in the provided direction.
public void moveAgent(Direction direction)
```

```
{
        int x = getXCoord(agentPosition);
        int y = getYCoord(agentPosition);

        // Swap the agent and the block based on the direction specified.
        switch (direction)
        {
                case LEFT:
                        if (x != 0 && grid[y][x - 1] != C)
                        {
                                setBlockPosition(grid[y][x - 1], getXCoord(
                                    ↪ agentPosition), getYCoord(agentPosition
                                    ↪ ));
                                setAgentPosition(x - 1, y);
                                lastDirection = Direction.LEFT;
                        }
                        break;
                case RIGHT:
                        if (x != size - 1 && grid[y][x + 1] != C)
                        {
                                setBlockPosition(grid[y][x + 1], getXCoord(
                                    ↪ agentPosition), getYCoord(agentPosition
                                    ↪ ));
                                setAgentPosition(x + 1, y);
                                lastDirection = Direction.RIGHT;
                        }
                        break;
                case UP:
                        if (y != 0 && grid[y - 1][x] != C)
                        {
                                setBlockPosition(grid[y - 1][x], getXCoord(
                                    ↪ agentPosition), getYCoord(agentPosition
                                    ↪ ));
                                setAgentPosition(x, y - 1);
                                lastDirection = Direction.UP;
                        }
                        break;
                case DOWN:
                        if (y != size - 1 && grid[y + 1][x] != C)
                        {
                                setBlockPosition(grid[y + 1][x], getXCoord(
                                    ↪ agentPosition), getYCoord(agentPosition
                                    ↪ ));
                                setAgentPosition(x, y + 1);
                                lastDirection = Direction.DOWN;
                        }
                        break;
        }
```

```
	}

	public Direction getLastDirection()
	{
		return lastDirection;
	}

	// Method that returns a boolean to see if two Grid objects have the same
	    ↪ char[][] grid.
	public boolean equals(Grid solution)
	{
		boolean equal;

		for (int i = 0; i < size; i++)
		{
			for (int j = 0; j < size; j++)
			{
				equal = grid[i][j] == solution.grid[i][j];

				if (!equal && grid[i][j] != A && grid[i][j] != C)
					return false;
			}
		}

		return true;
	}

	// Method that adds a collision block at the given x and y co-ordinates.
	private void addCollisionBlock(int x, int y)
	{
		int position = x + (size * y);

		if (blockPositions.containsValue(position) || getAgentPosition() ==
		    ↪ position)
			System.out.println("Cannot␣add␣collision␣block␣to␣co-
			    ↪ ordinate␣(" + x + ",␣" + ")␣as␣it␣is␣already␣occupied␣
			    ↪ by␣a␣block␣or␣the␣agent!");
		else
			grid[y][x] = '!';
	}

	public double calculateDistance(Grid solution)
	{
		double distance = 0;

		for (Map.Entry entry : blockPositions.entrySet())
		{
			int gridPosition = (int) entry.getValue();
```

```
                        int solutionPosition = solution.getBlockPosition((char)
                            ↪ entry.getKey());

                int gridX = getXCoord(gridPosition);
                int gridY = getYCoord(gridPosition);
                int solutionX = getXCoord(solutionPosition);
                int solutionY = getYCoord(solutionPosition);

                if (EUCLID)
                        distance = distance + calculateEuclideanDistance(gridX,
                            ↪ gridY, solutionX, solutionY);
                else
                distance = distance + calculateManhattanDistance(gridX, gridY,
                    ↪ solutionX, solutionY);
                }

                return distance;
        }

        // Function that calculates the Manhattan Distance between a position on
            ↪ this grid and on the solution grid.
        private int calculateManhattanDistance(int gridX, int gridY, int solutionX,
            ↪  int solutionY)
        {
                return Math.abs(gridX - solutionX) + Math.abs(gridY - solutionY);
        }

        // Function that calculates the Euclidean Distance between a position on
            ↪ this grid and on the solution grid.
        private double calculateEuclideanDistance(int gridX, int gridY, int
            ↪ solutionX, int solutionY)
        {
                return Math.sqrt(Math.pow((gridX - solutionX), 2) + Math.pow((gridY
                    ↪ - solutionY), 2));
        }
}
```

B.3 Direction

```
package Tree;

public enum Direction
{
        LEFT,
        RIGHT,
        UP,
        DOWN
}
```

B.4 Search

```
package Search;

import Tree.*;

import java.util.HashMap;
import java.util.Map;

// Abstract search class that defines a common start and solution state, and
    ↪ provides a common expandNode() method.
public abstract class Search
{
        private static final int N = 4;

        final Grid startState = new Grid(N);
        final Grid solutionState = new Grid(N);
        private int nodesGenerated = 1;
        private int fringeCount = 1;
        final boolean debugMode;

        Search(boolean debugMode)
        {
                this.debugMode = debugMode;
                solutionState.generateSolutionGrid();
        }

        public void setStartingGrid(int depth)
        {
                startState.generateStartingGrid(depth);
        }

        public Grid getStartingGrid()
        {
                return startState;
        }

        public int getNodesGenerated()
        {
                return nodesGenerated;
        }

        // Function that expands the given node by generating every possible
            ↪ successor node, i.e. the directions in which the agent can move.
        void expandNode(Node node)
        {
                Grid currentGrid = node.getValue();
```

```
            if (debugMode)
            {
                System.out.println("-------------␣Fringe␣" + fringeCount + "
                    ↪ ␣---------------\n");
                System.out.println("Selected␣node␣(depth␣" + node.getDepth()
                    ↪  + "):\n" + node.getValue() + "\n
                    ↪ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
            }

            // For each direction, attempt to move the agent in that direction
            for (Direction dir : Direction.values())
            {
                Map<Character, Integer> blockPositions = new HashMap<>();

                for (Map.Entry entry : currentGrid.getBlockPositions().
                    ↪ entrySet())
                {
                    char block = (char)entry.getKey();
                    int position = (int)entry.getValue();

                    blockPositions.put(block, position);
                }

                Grid newGrid = new Grid(currentGrid.getGrid(), currentGrid.
                    ↪ getAgentPosition(), blockPositions);

                newGrid.moveAgent(dir);

                if (!newGrid.equals(currentGrid))
                {
                    Node newNode = new Node(newGrid, node);
                    node.addChild(newNode);

                    nodesGenerated++;

                    if (debugMode)
                        System.out.println(newNode.getValue());
                }
            }

            fringeCount++;
        }

        Node returnSolution(Node node)
        {
            if (debugMode)
            {
```

```
                    System.out.println("---------␣Solution␣Node␣(depth␣" + node.
                        ↪ getDepth() + ")␣---------\n");
                    System.out.println(node.getValue());
                    System.out.println("
                        ↪ ------------------------------------------\n");
                }

                return node;
        }

        public abstract Node search();
}
```

B.5 BFS

```
package Search;

import java.util.LinkedList;
import java.util.Queue;
import Tree.Node;

// Class that implements the Breadth First Search (BFS) algorithm by using a Queue
    ↪ to manage the fringe.
public class BFS extends Search
{
	private final Queue<Node> fringe = new LinkedList<>();

	public BFS(boolean debugMode)
	{
		super(debugMode);
	}

	public Node search()
	{
		Node node = new Node(startState, null);

		fringe.add(node);

		while (!fringe.isEmpty())
		{
			node = fringe.remove();

			// Return the node if it is the solution state.
			if (node.getValue().equals(solutionState))
				return returnSolution(node);

			expandNode(node);

			// Add all the children to the queue, so they 'join the back
			    ↪ of the queue'.
			fringe.addAll(node.getChildren());
		}

		return null;
	}
}
```

B.6 DFS

```
package Search;

import Tree.Node;

import java.util.Collections;
import java.util.List;
import java.util.Stack;

// Class that implements the Depth First Search (DFS) algorithm by using a Stack
    ↪ to manage the fringe.
public class DFS extends Search
{
        private final Stack<Node> fringe = new Stack<>();
        private final boolean randomOrder;

        public DFS(boolean randomOrder, boolean debugMode)
        {
                super(debugMode);
                this.randomOrder = randomOrder;
        }

        public Node search()
        {
                Node node = new Node(startState, null);

                fringe.push(node);

                while (!fringe.isEmpty())
                {
                        node = fringe.pop();

                        // Return the node if it is the solution state.
                        if (node.getValue().equals(solutionState))
                                return returnSolution(node);

                        expandNode(node);

                        List<Node> children = node.getChildren();

                        // If the order of nodes is supposed to be random, call
                            ↪ Collections.shuffle on the children ArrayList.
                        if (randomOrder)
                                Collections.shuffle(children);

                        // Add all of the children to the Stack, so they are the
                            ↪ next nodes to be checked.
```

```
                        fringe.addAll(children);
                }

                return null;
        }
}
```

B.7 IDS

```
package Search;

import Tree.Node;

import java.util.List;
import java.util.Stack;

// Class that implements the Iterative Deepening Search (IDS) algorithm by using a
    ↪ Depth Limited Search (DLS), a modified version of DFS that stops at a
    ↪ certain limit, and then iterates this algorithm until a solution is found.
public class IDS extends Search
{
	private final Stack<Node> fringe = new Stack<>();

	public IDS(boolean debugMode)
	{
		super(debugMode);
	}

	public Node search()
	{
		Node solution = null;
		int i = 1;

		// While the solution has not been returned by DLS, increase the
		    ↪ limit by 1.
		while (solution == null)
		{
			solution = depthLimitedSearch(i);
			i++;
		}

		return solution;
	}

	private Node depthLimitedSearch(int limit)
	{
		Node node = new Node(startState, null);

		fringe.push(node);

		while (!fringe.isEmpty())
		{
			node = fringe.pop();

			// Return the node if it is the solution state.
```

```
                    if (node.getValue().equals(solutionState))
                        return returnSolution(node);

                    // If the limit has been reached, then do not expand this
                        ↪ node.
                    if (node.getDepth() == limit)
                        continue;

                    expandNode(node);

                    List<Node> children = node.getChildren();

                    // As with DFS, add all of the children to the Stack, so
                        ↪ they are the next nodes to be checked.
                    fringe.addAll(children);
                }

                return null;
        }
}
```

B.8 AStar

```
package Search;

import Tree.Node;

import java.util.PriorityQueue;
import java.util.Queue;

// Class that implements the A* Heuristic Search algorithm by using a
    ↪ PriorityQueue that prioritises nodes by their heuristic.
public class AStar extends Search
{
        private final Queue<Node> fringe = new PriorityQueue<>();

        public AStar(boolean debugMode)
        {
                super(debugMode);
        }

        public Node search()
        {
                Node node = new Node(startState, null);

                // Get the estimated cost of the node - the Manhattan distance (
                    ↪ heuristic) between the grid state and that of the solution.
                double estimatedCost = node.getValue().calculateDistance(
                    ↪ solutionState);
                node.setEstimatedCost(estimatedCost);

                fringe.add(node);

                while (!fringe.isEmpty())
                {
                        node = fringe.remove();

                        // Return the node if it is the solution state.
                        if (node.getValue().equals(solutionState))
                                return returnSolution(node);

                        expandNode(node);

                        int i = 1;

                        if (debugMode)
                        {
                                System.out.println("=====␣A*␣Heuristic␣Search␣
                                    ↪ information␣=====\n");
```

```
                        System.out.println("Depth at " + (node.getDepth() +
                            ↪ 1) + "\n");
                    }

                    // For each expanded child node, calculate the estimated
                        ↪ cost using the current depth and Manhattan distance (
                        ↪ heuristic), and put it into the PriorityQueue.
                    for (Node childNode : node.getChildren())
                    {
                        double heuristic = childNode.getValue().
                            ↪ calculateDistance(solutionState);
                        estimatedCost = childNode.getDepth() + heuristic;

                        if (debugMode)
                        {
                            System.out.println("Child " + i + ":");
                            System.out.println("Heuristic: " + heuristic);
                        System.out.println("Evaluation function value: " +
                            ↪ estimatedCost + "\n");
                        }

                        childNode.setEstimatedCost(estimatedCost);

                        fringe.add(childNode);

                        i++;
                    }
                }

                return null;
            }
}
```