COMP2208 - Search Methods Coursework

Daniel Best (Student ID: 29777127)

November 26, 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 to 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**, which then iteratively increases this limit.

1.7 A* Heuristic Search

Makes use of an evaluation function to determine which node to pick next, which is 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 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.

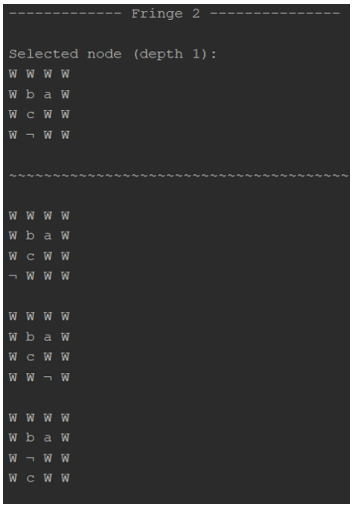
2.1 Breadth First Search (BFS)

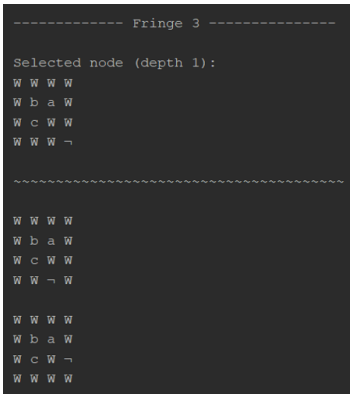
Example 1 (depth 4):

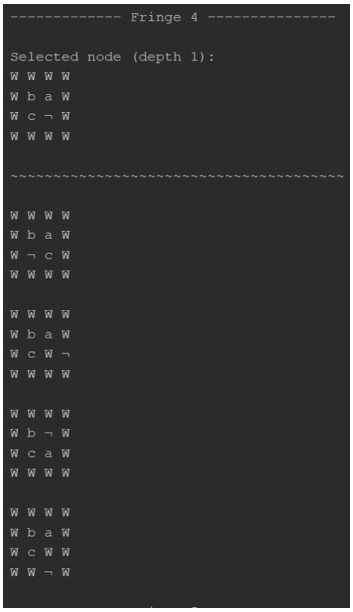
In this initial state, the agent has three possible directions that it can move.

```
********************************************************************************
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
```

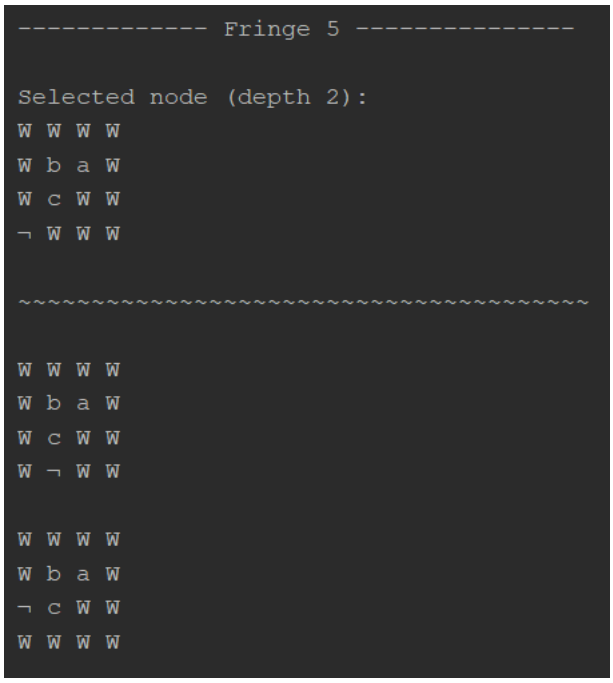
These three new nodes are then expanded in the order that they were first generated.



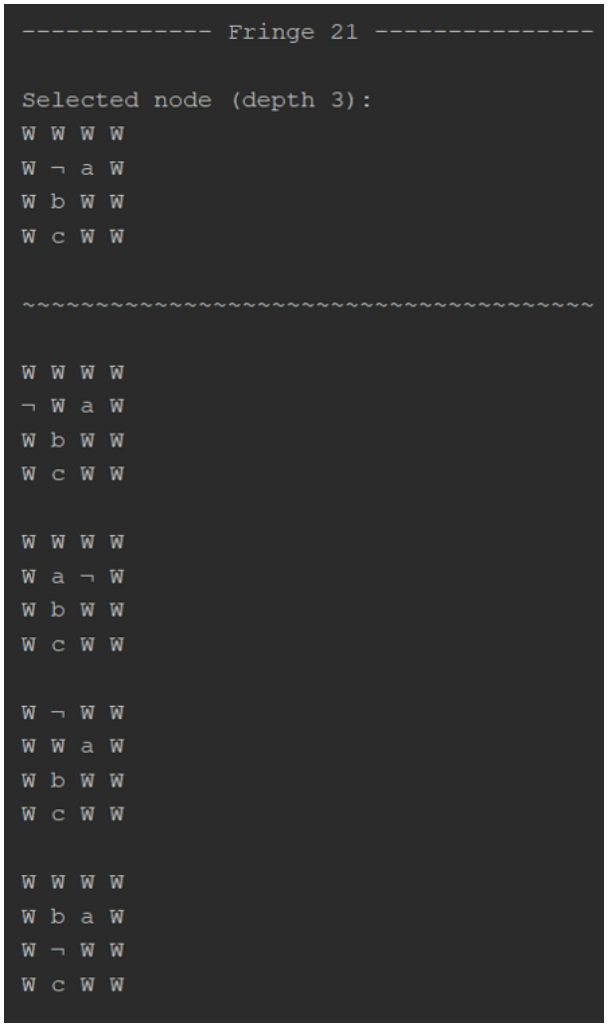




The algorithm then expands the first child node in **Fringe 2**, which generates the first nodes that are at a depth of 2.



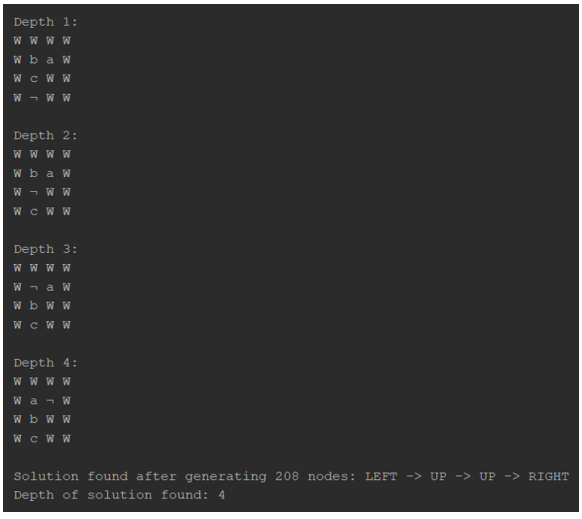
This process then repeats indefinitely until the correct node is found. At the 21st expansion to the fringe, we can see the correct node being found.



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 exist 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.

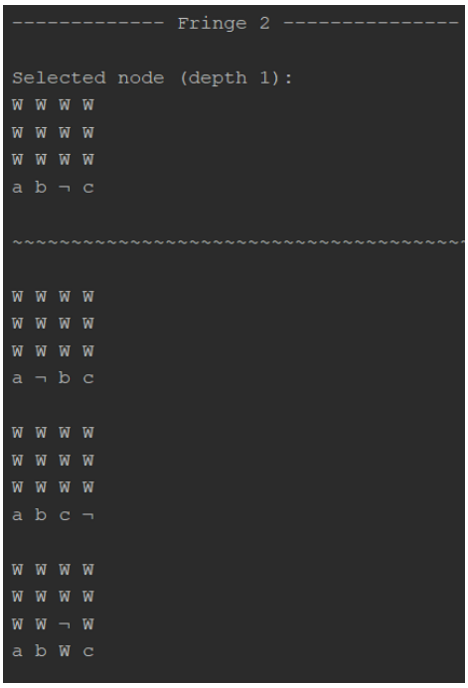


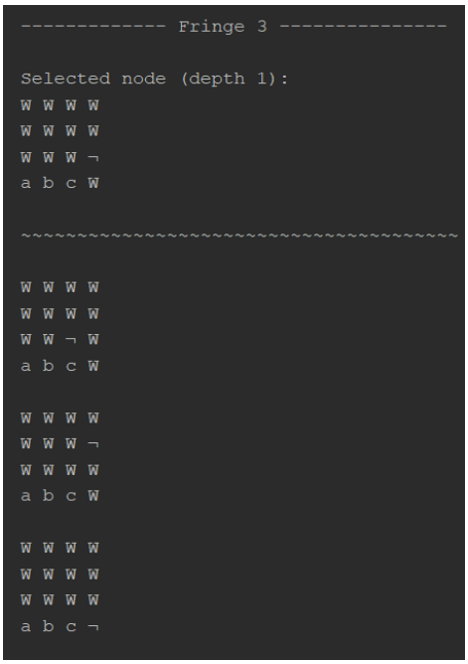
Example 2 (depth 14):

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.

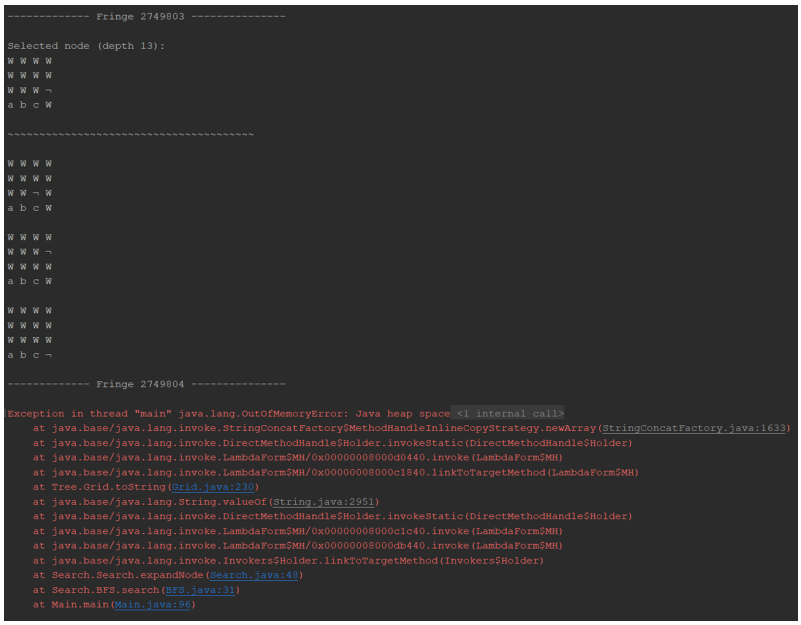
```
********************************************************************************
Breadth First Search (BFS) - Optimal Solution Depth: 14
********************************************************************************

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
```





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.

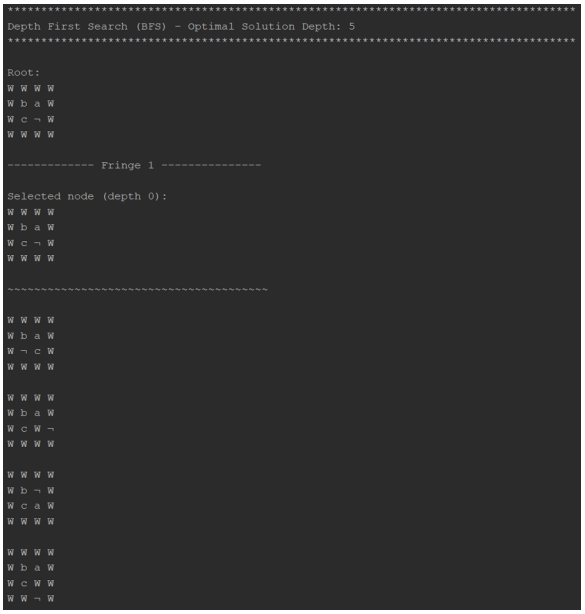
Eventually however, it will run into an **OutOfMemoryException** due to running out of Java Heap space.

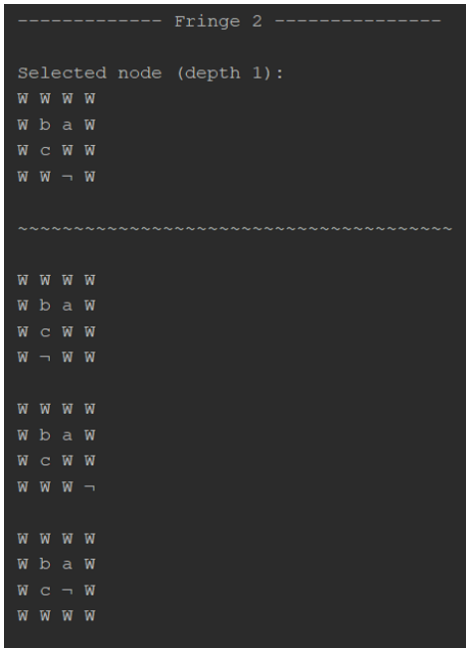


2.2 Depth First Search (DFS)

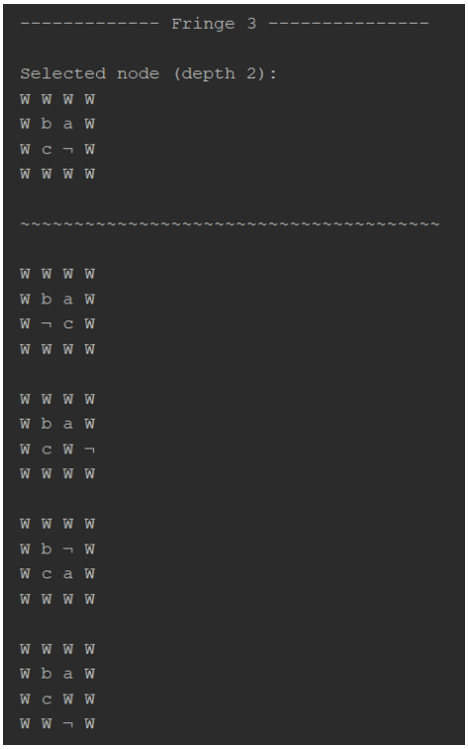
Example 1 (depth 5):

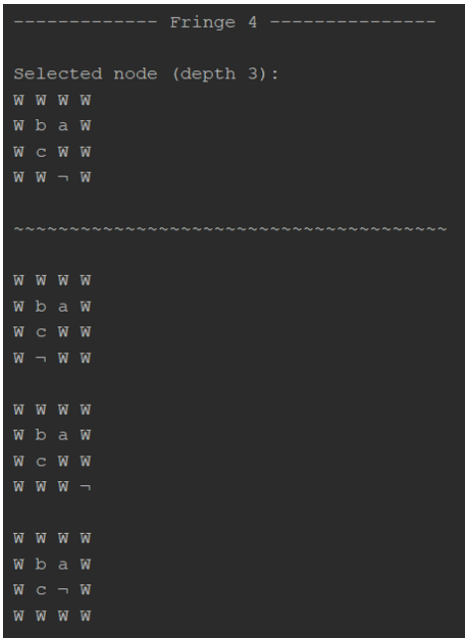
Implementing DFS for this type of problem has an obvious flaw in that the nodes are generated in an ordered manner. In the case of my algorithm, this is left, right, up and down. As such, the node added to the top of the **Stack** will always be the last possible direction. In this particular example,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



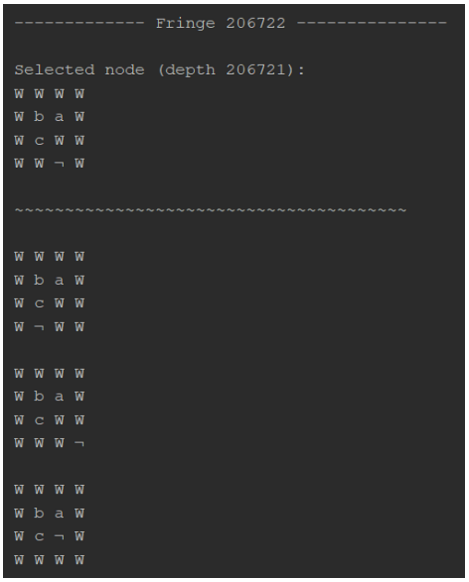


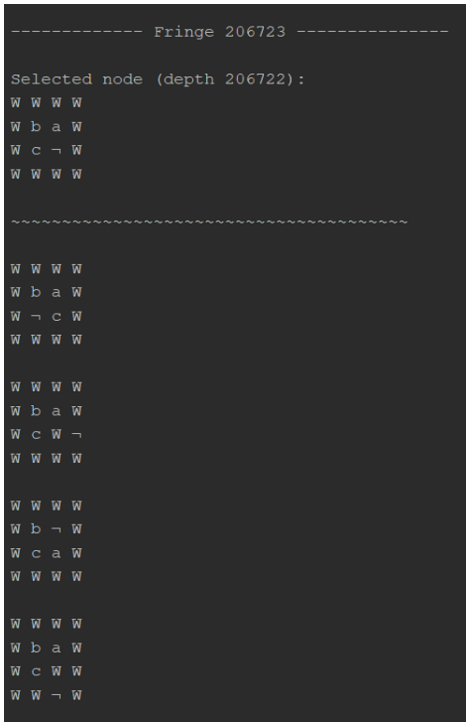
This means that unless modified accordingly, i.e. so the order of the generated nodes is randomised, the algorithm will almost certainly never find a solution.





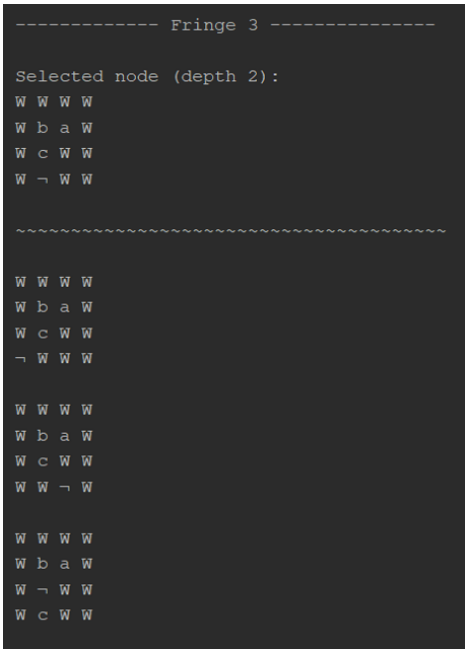
Eventually, this will cause an **OutOfMemoryException**, since it will repeat this process indefinitely, with the only difference being that the depth has increased by one each time.

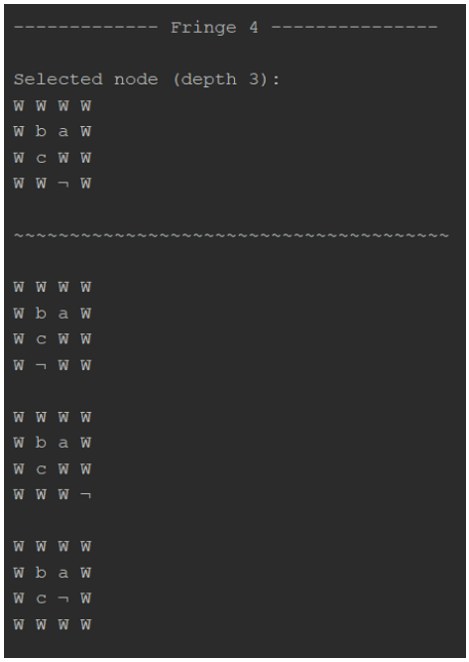




Example 2 (depth 5 with random order):

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.





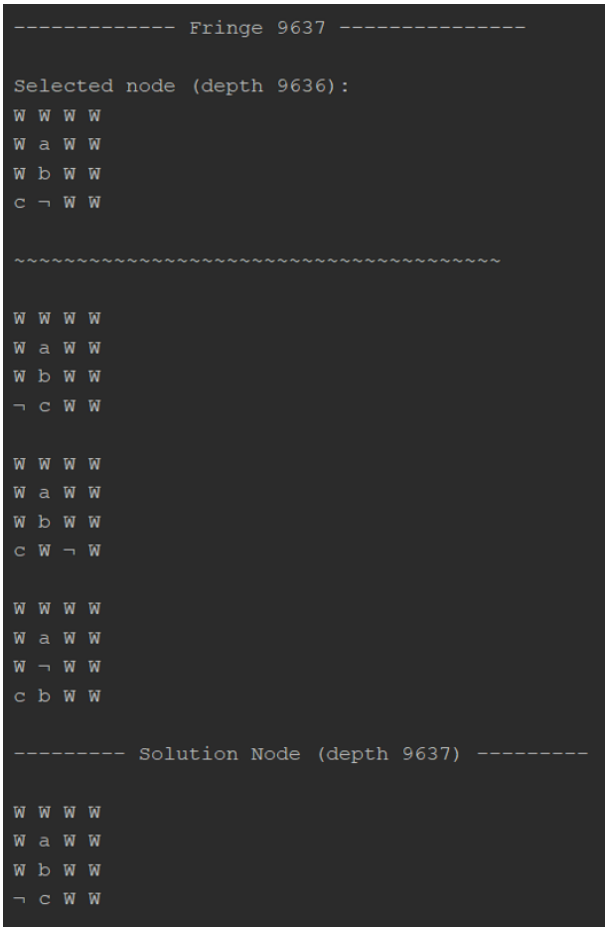
By completely arbitrary movements, the algorithm will eventually find some solution.

```
------------- Fringe 10967 ---------------

Selected node (depth 10966):
W W W W
a ¬ W W
W b W W
W c W W

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

W W W W
¬ a W W
W b W W
W c W W

W W W W
a W ¬ W
W b W W
W c W W

W ¬ W W
a W W W
W b W W
W c W W

W W W W
a b W W
W ¬ W W
W c W W

--------- Solution Node (depth 10967) ---------

W W W W
¬ a W W
W b W W
W c W W
```

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.

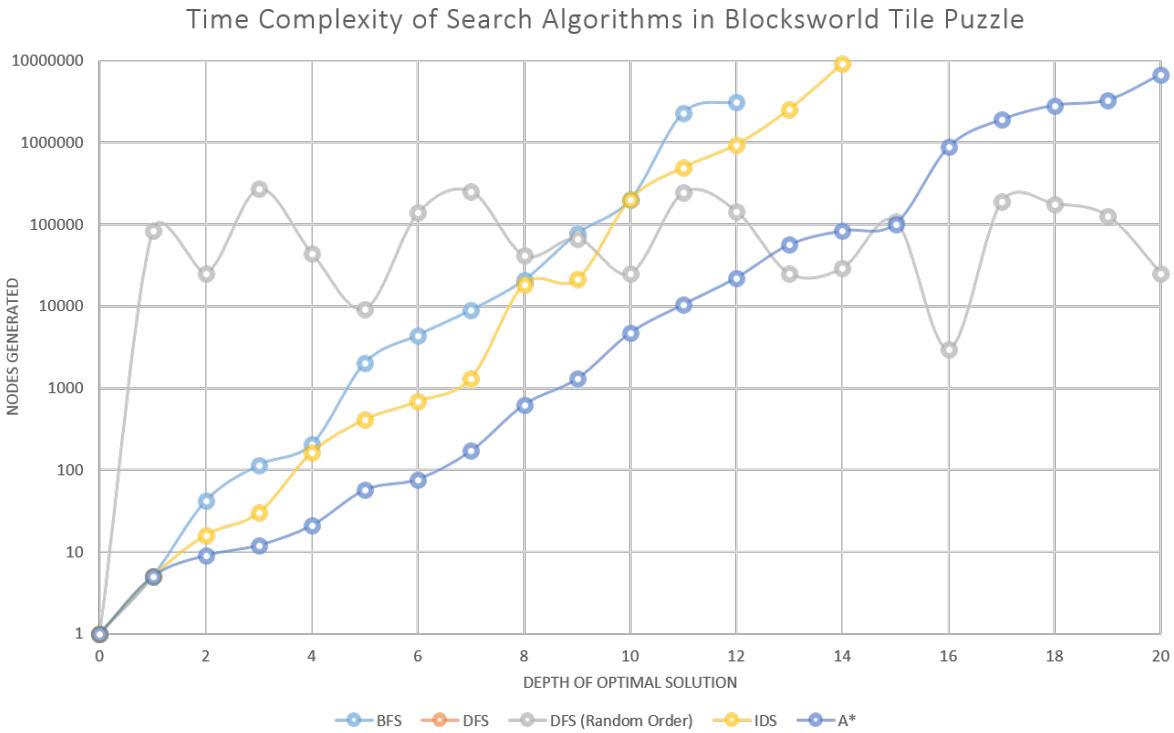


2.3 Iterative Deepening Search (IDS)

Example 1 (depth 6):

2.4 A* Heuristic Search

3 Scalability Study



4 Extras and Limitations

5 Code