COMP2208 - Intelligent Systems
Assignment: Search Methods

Shafiullah Rahman
Username: sr6g18
ID: 30556228

25th November 2019

1 Approach

I implemented the search methods in Java as it is an object oriented language. It is also the language I am most familiar with and where I can model a Node with ease.

I modelled a node by creating a “Node” class that stores a “State” object and the path to arrive at that node. It also contains various other methods that allow us to branch or check if the goal state has been achieved. The “State” object represents the state as a 2D array of strings where ‘0’ represents a blank tile, ‘A’, ‘B’, ‘C’ represent the blocks that require moving and ‘S’ identifies the agent.

The path is a string of Us, Ds, Ls, Rs that identify Up, Down, Left and Right respectively.

1.1 Breadth First Search

Implemented a queue with a LinkedList to store the fringe nodes. The method goes through each node in the queue checking if the node is the goal state before branching and adding the generated nodes to the end of the queue. If it is the goal state, we will print the path.

1.2 Depth First Search

Used a stack to store the fringe nodes. The method pops the node off of the top of the stack checking if the node is the goal state before branching and adding the generated nodes to the top of the stack. The generated nodes are added in a random order so that node expansion is random when a node is popped to avoid an infinite run time.

1.3 Iterative Deepening Search

Used a stack to store the fringe nodes. The method pops the node off of the top of the stack checking if the node is the goal state and then checking if the depth of the node is less than the depth limit before branching and adding the generated nodes to the top of the stack. If the depth of the node is equal to the depth limit then we go through the current stack and check if any of the nodes are the goal state. If the stack is empty and no goal state has been found then we reset the depth limited search and increment depth limit by one.

1.4 A* Heuristic Search

Uses a priority queue to store the fringe nodes. The method goes through each node in the queue checking if the node is the goal state before branching and adding the generated nodes to the priority queue. Our heuristic in this case is the distance from the start positions of A, B and C to the goal positions of A,B and C. We have a method called “getHeuristic” that calculates this and “compareTo” which compares the nodes in the priority queue on the sum of their depth and heuristic value. Nodes with the smallest sum are the first to be removed.

1.5 Other

We modelled the search such that the agent is truly uninformed so that it can still move towards a wall but it will not go outside of the 4x4 grid. This means the agent can move in any direction it wants within the 4x4 grid.

2 Evidence

2.1 Breadth First Search

As the root node expands, the following output is observed. Since there are many lines of output, I am showing the initial few and "...", followed by the last few to indicate all lines of output between. The BFS is space intensive so the search is unable to complete as we are out of heap space memory hence the "OutOfMemoryError".

```
Nodes Generated: 0 -- Path: -- Depth: 0
Nodes Generated: 4 -- Path: U -- Depth: 1
Nodes Generated: 8 -- Path: D -- Depth: 1
Nodes Generated: 12 -- Path: L -- Depth: 1
Nodes Generated: 16 -- Path: R -- Depth: 1
Nodes Generated: 20 -- Path: UU -- Depth: 2
Nodes Generated: 24 -- Path: UD -- Depth: 2
...
Nodes Generated: 18346372 -- Path: RUULLDRUURU -- Depth: 11
Nodes Generated: 18346376 -- Path: RUULLDRUURD -- Depth: 11
Nodes Generated: 18346380 -- Path: RUULLDRUURL -- Depth: 11
Nodes Generated: 18346384 -- Path: RUULLDRUURR -- Depth: 11
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
   at Node.nodeDown(Node.java:93)
   at Node.branch(Node.java:49)
   at Main.breadthFirst(Main.java:24)
   at Main.main(Main.java:10)
```

2.2 Depth First Search

As before, most lines of output have been omitted as there are too many. The program successfully finds a solution this time. Since it is a DFS, it sacrifices completeness for space efficiency. The path is substantially longer than other search methods yielding a length of 5988. Since the path is so long I have placed "'..."', to indicate the rest of the path.

```
Nodes Generated: 0 -- Path: -- Depth: 0
Nodes Generated: 4 -- Path: U -- Depth: 1
Nodes Generated: 8 -- Path: UR -- Depth: 2
Nodes Generated: 12 -- Path: URU -- Depth: 3
Nodes Generated: 16 -- Path: URUD -- Depth: 4
Nodes Generated: 20 -- Path: URUDU -- Depth: 5
Nodes Generated: 24 -- Path: URUDUL -- Depth: 6
...
Nodes Generated: 23940 -- Path: RLDRURLU ... DRLUDLUR -- Depth: 5985
Nodes Generated: 23944 -- Path: RLDRURLU ... RLUDLURU -- Depth: 5986
Nodes Generated: 23948 -- Path: RLDRURLU ... LUDLURUL -- Depth: 5987
Nodes Generated: 23952 -- Path: RLDRURLU ... UDLURULL -- Depth: 5988

Goal State Found:
Path: RLDRURLU ... UDLURULL
Nodes Generated: 23952
Depth: 5988
```

2.3 Iterative Deepening Search

As before, most lines of output have been omitted as there are too many. This program successfully finds a solution too, however, it generates significantly more nodes than the previous methods at 420 million. We are essentially combining DFS's space efficiency with BFS's completeness. It is therefore much slower than DFS. The path is shorter since it is complete due to the depth limit and since we are not randomising node expansion as we did in the DFS.

```
Nodes Generated: 0 -- Path: -- Depth: 0 -- Depth Limit: 0
Nodes Generated: 4 -- Path: R -- Depth: 1 -- Depth Limit: 1
Nodes Generated: 4 -- Path: L -- Depth: 1 -- Depth Limit: 1
Nodes Generated: 4 -- Path: D -- Depth: 1 -- Depth Limit: 1
Nodes Generated: 4 -- Path: U -- Depth: 1 -- Depth Limit: 1
Nodes Generated: 8 -- Path: R -- Depth: 1 -- Depth Limit: 2
Nodes Generated: 12 -- Path: RR -- Depth: 2 -- Depth Limit: 2
...
Nodes Generated: 419122240 -- Path: ULLDLURDRUULLU -- Depth: 14 -- Depth Limit: 14
Nodes Generated: 419122240 -- Path: ULLDLURDRUULD -- Depth: 13 -- Depth Limit: 14
Nodes Generated: 419122244 -- Path: ULLDLURDRUULDR -- Depth: 14 -- Depth Limit: 14
Nodes Generated: 419122244 -- Path: ULLDLURDRUULDL -- Depth: 14 -- Depth Limit: 14

Goal State Found:
Path: ULLDLURDRUULDL
Nodes Generated: 419122244
Depth: 14
Depth Limit: 14
```

2.4 A* Search

As before, most lines of output have been omitted as there are too many. This program is successful too and is significantly more efficient than previous methods generating about one million nodes and an optimal solution at depth 14 due to the use of a heuristic. It provides the same solution as iterative deepening, except that it works a lot faster generating only one million nodes as opposed to 400 million nodes.

```
Nodes Generated: 0 -- Path: -- Depth: 0
Nodes Generated: 4 -- Path: U -- Depth: 1
Nodes Generated: 8 -- Path: R -- Depth: 1
Nodes Generated: 12 -- Path: D -- Depth: 1
Nodes Generated: 16 -- Path: RR -- Depth: 2
Nodes Generated: 20 -- Path: DR -- Depth: 2
...
Nodes Generated: 1063668 -- Path: ULRLLRUDR -- Depth: 9
Nodes Generated: 1063672 -- Path: ULRLLRULU -- Depth: 9
Nodes Generated: 1063676 -- Path: ULRLLRUDLD -- Depth: 10
Nodes Generated: 1063680 -- Path: ULLDLURDRUULDL -- Depth: 14

Goal State Found:
Path: ULLDLURDRUULDL
Nodes Generated: 1063680
Depth: 14
```

3 Scalability Study

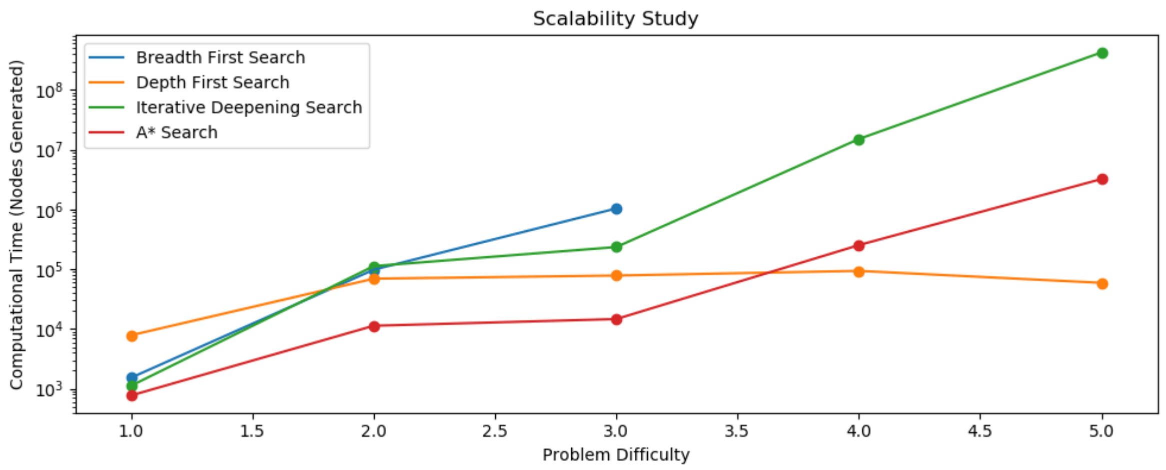
3.1 Problem Difficulty

To control problem difficulty, I will move the positions of A, B and C. To increase difficulty: move ‘A’, ‘B’ and ‘C’ further away from their goal positions. To decrease difficulty: move ‘A’, ‘B’ and C closer to their goal positions. Our difficulty is determined by the sum of the moves required for ‘A’, ‘B’ and ‘C’ to move to their correct positions. A higher number of moves means a higher difficulty and vice versa.

For the default start state of the problem, we have a problem difficulty of 5 since ‘A’ is 3 moves away from its goal position (2 Up + 1 Right), ‘B’ is 1 move away (1 Up) and ‘C’ is 1 move away (1 left). The sum of these moves equals 5. This is similar to our heuristic in the A* Search.

On the other hand, the goal state of the problem has a problem difficulty of 0 since ‘A’, ‘B’ and ‘C’ require 0 moves to get into their correct positions.

3.2 Scalability Plot



3.3 Interpretation

As the problem difficulty rises, the computational time of DFS remains stagnant. This is because for every increase in problem difficulty, the depth is raised and only the minimum traversal to a goal state is increased meaning that it has no substantial effect on the average. DFS has a time complexity of $0(4^m)$ where m is the maximum depth of the state space (∞).

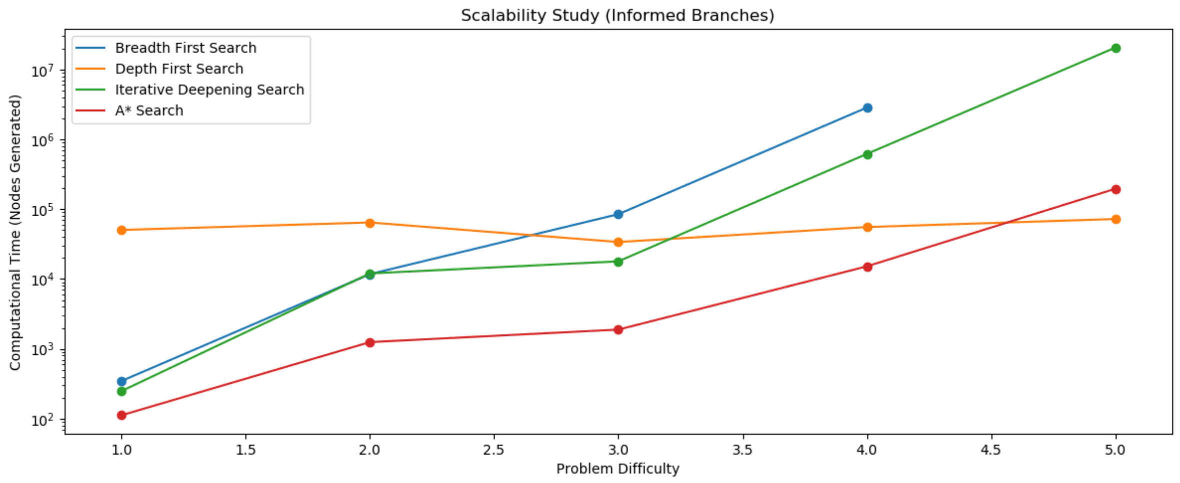
On the other hand, BFS and IDS remain similar in computational time as both appear to increase rapidly due to their exhaustive checking for optimal solutions. BFS however stops at difficulty 4 since we run out of heap space memory. IDS, however, offers significantly better space complexity. BFS has a time complexity of $O(4^{d+1})$ and IDS a complexity of $O(4^d)$ where d = depth of the optimal solution.

A* Search has a substantially smaller computational time as opposed to IDS and BFS since the heuristic allows the order of node expansion to be meaningful, not exhaustive as it it in IDS and BFS. A* has a time complexity of $O(4^d)$ but better as the heuristic has an effect on the performance. Although the exhaustive nature of IDS and BFS allow them to find optimal solutions, A* sacrifices optimality for a significant increase in space complexity. The graph suggests that A* is slower than DFS, however due to the random order of node expansion in our DFS, it is unfair to say it is quicker than A*. Moreover, DFS lacks completeness as it is possible for it to loop indefinitely.

4 Extras and Limitations

4.1 Informed Branches

In our previous searches the agent is able to move towards the wall. For example, if the agent is in its default position in the bottom right then it will be uninformed since it can move Up, Down, Left or Right. Moving Down or Right will make the agent stay in that position since it cannot move out of the grid. By changing the way we branch, we can make the search less uninformed so that it does not waste moves going towards the wall. After this change the agent will not be able to move towards the outside meaning it is only able to move Up and Left at the default position for example. We can now run our methods with this less uninformed search and expect to see a fall in computational time.



This graph shows that the search methods perform significantly better when the agent branches correctly at the edges of the grid. BFS found a solution at a problem difficulty of 4, without incurring the Java "OutOfMemoryError" it did without informed branching. DFS maintained its computational time as a decrease in depth has no substantial impact due to the randomness of the node expansion. IDS is very similar to before except that it has also seen vast improvements in performance and efficiency. So too has the A* search. All the methods have benefited from informed branches both in terms of time complexity (the no. of nodes generated) and space complexity(the no. of nodes stored in memory), due to the introduction of a smaller branching factor when agents are next to a wall.