Search Methods

Approach

To complete the search methods, I wrote the programs necessary in a little known and little used language: Java. The object oriented nature of the language meant that I initially tried hard to make my solution follow the four pillars of good programming. However, I soon found out that this coursework was actually rather difficult and perhaps object oriented programming was unsuited to this task due to its sometimes baffling system of object references getting in the way. Therefore, I abandoned all principles and just put everything in one class and used lots of static methods like an uneducated clown.

All the necessary code to run the searches is found in two classes: Main.java, which holds the search methods, and Node.java, which stores the grid and models a node of a search algorithm. The grid is stored as a 2D integer array where: 0 represents an empty square, 1 represents the A block, 2 represents the B block, 3 represents the C block, and 9 represents the agent. Also stored in the node is the agent's x position, the agent's y position, and the last move done by the agent.

The node class has public methods for getting the node's depth, getting the node's parent, finding the node's children, and getting the node's last move as well as a method to print the status of the grid and a method to check if the grid is complete.

Evidence

Depth First Search

Due to the nature of both the puzzle and how this search method was implemented, depth first search evidence can best be described as the completely random movement of the agent on the miniscule hope that the algorithm might blunder its way into success. Despite my lack of hope, solutions can be found using this search with a wide variety of search times. One solution was found at a depth of 102'688, another at a depth of 33'262, and another at a depth of 61'204. The last 20 iterations of output for that last one is detailed below, showing the depth first search in action:

Note: my solution abstracts the problem by representing the grid in the Blocksworld game as a 2D array of integers where 0 represents an empty square, 1 represents the A block, 2 represents the B block, 3 represents the C block, and 9 represents the roaming agent. This raw output should be considered with this in mind.

Step: 61185. Fringe size: 132338. Depth: 61185. Last move: Left. --- | 0 | 0 | 0 | 0 | --- | 0 | 1 | 0 | 0 | --- | 0 | 9 | 2 | 0 | --- | 0 | 3 | 0 | 0 | --- Adding next level to fringe	Step: 61186. Fringe size: 132341. Depth: 61186. Last move: Left. --- | 0 | 0 | 0 | 0 | --- | 0 | 1 | 0 | 0 | --- | 9 | 0 | 2 | 0 | --- | 0 | 3 | 0 | 0 | --- Adding next level to fringe	Step: 61187. Fringe size: 132343. Depth: 61187. Last move: Down. --- | 0 | 0 | 0 | 0 | --- | 0 | 1 | 0 | 0 | --- | 0 | 0 | 2 | 0 | --- | 9 | 3 | 0 | 0 | --- Adding next level to fringe	Step: 61188. Fringe size: 132344. Depth: 61188. Last move: Right. --- | 0 | 0 | 0 | 0 | --- | 0 | 1 | 0 | 0 | --- | 0 | 0 | 2 | 0 | --- | 3 | 9 | 0 | 0 | --- Adding next level to fringe
Step: 61189. Fringe size: 132346. Depth: 61189. Last move: Right. --- | 0 | 0 | 0 | 0 | --- | 0 | 1 | 0 | 0 | --- | 0 | 0 | 2 | 0 | ---	Step: 61190. Fringe size: 132348. Depth: 61190. Last move: Right. --- | 0 | 0 | 0 | 0 | --- | 0 | 1 | 0 | 0 | --- | 0 | 0 | 2 | 0 | ---	Step: 61191. Fringe size: 132349. Depth: 61191. Last move: Left. --- | 0 | 0 | 0 | 0 | --- | 0 | 1 | 0 | 0 | --- | 0 | 0 | 2 | 0 | ---	Step: 61192. Fringe size: 132351. Depth: 61192. Last move: Up. --- | 0 | 0 | 0 | 0 | --- | 0 | 1 | 0 | 0 | --- | 0 | 0 | 9 | 0 | ---

| 3 | 0 | 9 | 0 | --- Adding next level to fringe	| 3 | 0 | 0 | 9 | --- Adding next level to fringe	| 3 | 0 | 9 | 0 | --- Adding next level to fringe	| 3 | 0 | 2 | 0 | --- Adding next level to fringe
Step: 61193. Fringe size: 132354. Depth: 61193. Last move: Down. --- | 0 | 0 | 0 | 0 | --- | 0 | 1 | 0 | 0 | --- | 0 | 0 | 2 | 0 | --- | 3 | 0 | 9 | 0 | --- Adding next level to fringe	Step: 61194. Fringe size: 132356. Depth: 61194. Last move: Right. --- | 0 | 0 | 0 | 0 | --- | 0 | 1 | 0 | 0 | --- | 0 | 0 | 2 | 0 | --- | 3 | 0 | 0 | 9 | --- Adding next level to fringe	Step: 61195. Fringe size: 132357. Depth: 61195. Last move: Left. --- | 0 | 0 | 0 | 0 | --- | 0 | 1 | 0 | 0 | --- | 0 | 0 | 2 | 0 | --- | 3 | 0 | 9 | 0 | --- Adding next level to fringe	Step: 61196. Fringe size: 132359. Depth: 61196. Last move: Left. --- | 0 | 0 | 0 | 0 | --- | 0 | 1 | 0 | 0 | --- | 0 | 0 | 2 | 0 | --- | 3 | 9 | 0 | 0 | --- Adding next level to fringe
Step: 61197. Fringe size: 132361. Depth: 61197. Last move: Left. --- | 0 | 0 | 0 | 0 | --- | 0 | 1 | 0 | 0 | --- | 0 | 0 | 2 | 0 | --- | 9 | 3 | 0 | 0 | --- Adding next level to fringe	Step: 61198. Fringe size: 132362. Depth: 61198. Last move: Up. --- | 0 | 0 | 0 | 0 | --- | 0 | 1 | 0 | 0 | --- | 9 | 0 | 2 | 0 | --- | 0 | 3 | 0 | 0 | --- Adding next level to fringe	Step: 61199. Fringe size: 132364. Depth: 61199. Last move: Up. --- | 0 | 0 | 0 | 0 | --- | 9 | 1 | 0 | 0 | --- | 0 | 0 | 2 | 0 | --- | 0 | 3 | 0 | 0 | --- Adding next level to fringe	Step: 61200. Fringe size: 132366. Depth: 61200. Last move: Down. --- | 0 | 0 | 0 | 0 | --- | 0 | 1 | 0 | 0 | --- | 9 | 0 | 2 | 0 | --- | 0 | 3 | 0 | 0 | --- Adding next level to fringe
Step: 61201. Fringe size: 132368. Depth: 61201. Last move: Right. --- | 0 | 0 | 0 | 0 | --- | 0 | 1 | 0 | 0 | --- | 0 | 9 | 2 | 0 | --- | 0 | 3 | 0 | 0 | --- Adding next level to fringe	Step: 61202. Fringe size: 132371. Depth: 61202. Last move: Right. --- | 0 | 0 | 0 | 0 | --- | 0 | 1 | 0 | 0 | --- | 0 | 2 | 9 | 0 | --- | 0 | 3 | 0 | 0 | --- Adding next level to fringe	Step: 61203. Fringe size: 132374. Depth: 61203. Last move: Down. --- | 0 | 0 | 0 | 0 | --- | 0 | 1 | 0 | 0 | --- | 0 | 2 | 0 | 0 | --- | 0 | 3 | 9 | 0 | --- Adding next level to fringe	Step: 61204. Fringe size: 132376. Depth: 61204. Last move: Right. --- | 0 | 0 | 0 | 0 | --- | 0 | 1 | 0 | 0 | --- | 0 | 2 | 0 | 0 | --- | 0 | 3 | 0 | 9 | --- Solution found

I will spare the reader the horror of knowing all 61'204 steps required to find this solution which, if typed with this font size and using one character per move, would fill 19 pages of this report.

Breadth First Search

This search method looks at all nodes at a certain depth before moving on to nodes with a greater depth. Therefore to show that this search is working as designed, the output would have to display the depth of the node, showing how the search tree examines the nodes in a specific order. To demonstrate that this search is working correctly, the first 20 steps' outputs are shown below, using the same notation style as the depth first search output:

Step: 0. Fringe size: 0. Depth: 0. Last move: null. --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 1 | 2 | 3 | 9 | --- Adding next level to fringe	Step: 1. Fringe size: 1. Depth: 1. Last move: Left. --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 1 | 2 | 9 | 3 | --- Adding next level to fringe	Step: 2. Fringe size: 3. Depth: 1. Last move: Up. --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 9 | --- | 1 | 2 | 3 | 0 | --- Adding next level to fringe	Step: 3. Fringe size: 5. Depth: 2. Last move: Right. --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 1 | 2 | 3 | 9 | --- Adding next level to fringe
Step: 4. Fringe size: 6. Depth: 2. Last move: Up. --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 9 | 0 | --- | 1 | 2 | 0 | 3 | --- Adding next level to fringe	Step: 5. Fringe size: 9. Depth: 2. Last move: Left. --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 1 | 9 | 2 | 3 | --- Adding next level to fringe	Step: 6. Fringe size: 11. Depth: 2. Last move: Down. --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 1 | 2 | 3 | 9 | --- Adding next level to fringe	Step: 7. Fringe size: 12. Depth: 2. Last move: Left. --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 9 | 0 | --- | 1 | 2 | 3 | 0 | --- Adding next level to fringe

Step: 8. Fringe size: 15. Depth: 2. Last move: Up. ---------- | 0 | 0 | 0 | 0 | ---------- | 0 | 0 | 0 | 9 | ---------- | 0 | 0 | 0 | 0 | ---------- | 1 | 2 | 3 | 0 | ---------- Adding next level to fringe	Step: 9. Fringe size: 17. Depth: 3. Last move: Up. ---------- | 0 | 0 | 0 | 0 | ---------- | 0 | 0 | 0 | 0 | ---------- | 0 | 0 | 0 | 9 | ---------- | 1 | 2 | 3 | 0 | ---------- Adding next level to fringe	Step: 10. Fringe size: 19. Depth: 3. Last move: Left. ---------- | 0 | 0 | 0 | 0 | ---------- | 0 | 0 | 0 | 0 | ---------- | 0 | 0 | 0 | 0 | ---------- | 1 | 2 | 9 | 3 | ---------- Adding next level to fringe	Step: 11. Fringe size: 21. Depth: 3. Last move: Down. ---------- | 0 | 0 | 0 | 0 | ---------- | 0 | 0 | 0 | 0 | ---------- | 0 | 0 | 0 | 0 | ---------- | 1 | 2 | 9 | 3 | ---------- Adding next level to fringe
Step: 12. Fringe size: 23. Depth: 3. Last move: Up. ---------- | 0 | 0 | 0 | 0 | ---------- | 0 | 0 | 9 | 0 | ---------- | 0 | 0 | 0 | 0 | ---------- | 1 | 2 | 0 | 3 | ---------- Adding next level to fringe	Step: 13. Fringe size: 26. Depth: 3. Last move: Right. ---------- | 0 | 0 | 0 | 0 | ---------- | 0 | 0 | 0 | 0 | ---------- | 0 | 0 | 0 | 9 | ---------- | 1 | 2 | 0 | 3 | ---------- Adding next level to fringe	Step: 14. Fringe size: 28. Depth: 3. Last move: Left. ---------- | 0 | 0 | 0 | 0 | ---------- | 0 | 0 | 0 | 0 | ---------- | 0 | 9 | 0 | 0 | ---------- | 1 | 2 | 0 | 3 | ---------- Adding next level to fringe	Step: 15. Fringe size: 31. Depth: 3. Last move: Right. ---------- | 0 | 0 | 0 | 0 | ---------- | 0 | 0 | 0 | 0 | ---------- | 0 | 0 | 0 | 0 | ---------- | 1 | 2 | 9 | 3 | ---------- Adding next level to fringe
Step: 16. Fringe size: 33. Depth: 3. Last move: Up. ---------- | 0 | 0 | 0 | 0 | ---------- | 0 | 0 | 0 | 0 | ---------- | 0 | 9 | 0 | 0 | ---------- | 1 | 0 | 2 | 3 | ---------- Adding next level to fringe	Step: 17. Fringe size: 36. Depth: 3. Last move: Left. ---------- | 0 | 0 | 0 | 0 | ---------- | 0 | 0 | 0 | 0 | ---------- | 0 | 0 | 0 | 0 | ---------- | 9 | 1 | 2 | 3 | ---------- Adding next level to fringe	Step: 18. Fringe size: 37. Depth: 3. Last move: Left. ---------- | 0 | 0 | 0 | 0 | ---------- | 0 | 0 | 0 | 0 | ---------- | 0 | 0 | 0 | 0 | ---------- | 1 | 2 | 9 | 3 | ---------- Adding next level to fringe	Step: 19. Fringe size: 39. Depth: 3. Last move: Up. ---------- | 0 | 0 | 0 | 0 | ---------- | 0 | 0 | 0 | 0 | ---------- | 0 | 0 | 0 | 9 | ---------- | 1 | 2 | 3 | 0 | ---------- Adding next level to fringe

Iterative Deepening

This search method looks at every node before a certain node depth before increasing the maximum depth to look at and restarting the search. For my solution, the output is a print of the depth to search up to thus confirming that the search depth is iteratively increasing until a solution is found.

Starting Iterative Deepening Search
Current search depth: 0
Current search depth: 1
Current search depth: 2
Current search depth: 3
Current search depth: 4
Current search depth: 5
Current search depth: 6
Current search depth: 7
Current search depth: 8
Current search depth: 9
Current search depth: 10
Current search depth: 11
Current search depth: 12
Current search depth: 13
Current search depth: 14
Current search depth: 15
Current search depth: 16
Solution node found!
Solution found at depth 16

| 0 | 0 | 0 | 0 |

| 0 | 1 | 0 | 0 |

| 0 | 2 | 0 | 0 |

| 0 | 3 | 0 | 9 |

Due to the reasonably low depth of this optimal solution, we can find the exact path the agent takes to beat the puzzle. This path is:

Up, Left, Down, Left, Left, Up, Right, Down, Right, Up, Up, Left, Down, Down, Right, Right

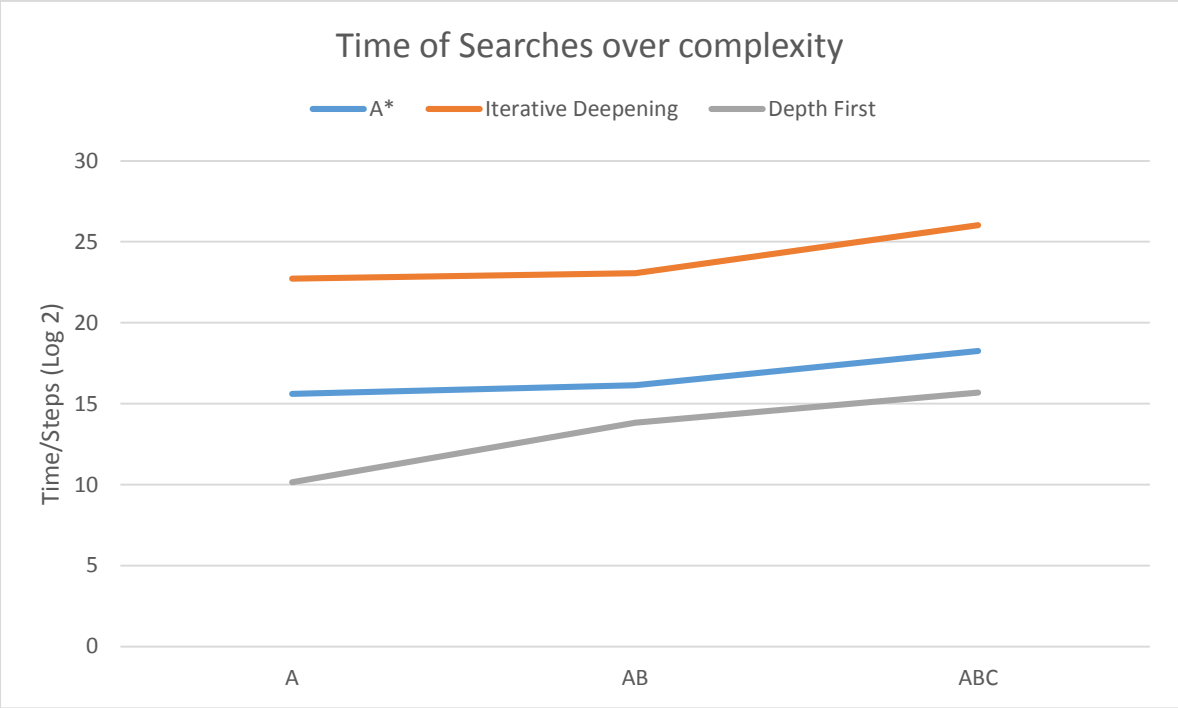
A* Heuristic Search

This search heuristic uses knowledge of the game to better search for the solution to the problem by calculating a 'score' assigned to each node and figuring out which nodes are worth expanding based on that score - nodes with lower scores will be expanded before those nodes with higher scores. The scores are the sum of the depth of the node and the Manhattan distance of the blocks from their current location to the complete final location. The verification of this by looking at the evidence is left as an exercise to the reader. My implementation of this heuristic found the same optimal solution at depth 16 as the iterative deepening search. To demonstrate the heuristic works as intended, the first 20 moves are shown below:

Step: 0. Fringe size: 0. Depth: 0. Last move: null. Score: 5 --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 1 | 2 | 3 | 9 | ---	Step: 1. Fringe size: 1. Depth: 1. Last move: Up. Score: 7 --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 9 | --- | 1 | 2 | 3 | 0 | ---	Step: 2. Fringe size: 3. Depth: 2. Last move: Down. Score: 7 --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 1 | 2 | 3 | 9 | ---	Step: 3. Fringe size: 4. Depth: 1. Last move: Left. Score: 8 --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 1 | 2 | 9 | 3 | ---
Step: 4. Fringe size: 6. Depth: 2. Last move: Right. Score: 7 --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 1 | 2 | 3 | 9 | ---	Step: 5. Fringe size: 7. Depth: 2. Last move: Left. Score: 9 --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 9 | 0 | --- | 1 | 2 | 3 | 0 | ---	Step: 6. Fringe size: 10. Depth: 3. Last move: Up. Score: 9 --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 9 | --- | 1 | 2 | 3 | 0 | ---	Step: 7. Fringe size: 12. Depth: 3. Last move: Up. Score: 9 --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 9 | --- | 1 | 2 | 3 | 0 | ---
Step: 8. Fringe size: 14. Depth: 3. Last move: Right. Score: 9 --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 9 | --- | 1 | 2 | 3 | 0 | ---	Step: 9. Fringe size: 16. Depth: 2. Last move: Up. Score: 9 --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 9 | --- | 0 | 0 | 0 | 0 | --- | 1 | 2 | 3 | 0 | ---	Step: 10. Fringe size: 18. Depth: 4. Last move: Down. Score: 9 --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 1 | 2 | 3 | 9 | ---	Step: 11. Fringe size: 19. Depth: 3. Last move: Down. Score: 9 --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 9 | --- | 1 | 2 | 3 | 0 | ---
Step: 12. Fringe size: 21. Depth: 4. Last move: Down. Score: 9 --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 1 | 2 | 3 | 9 | ---	Step: 13. Fringe size: 22. Depth: 4. Last move: Down. Score: 9 --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 1 | 2 | 3 | 9 | ---	Step: 14. Fringe size: 23. Depth: 4. Last move: Down. Score: 9 --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 1 | 2 | 3 | 9 | ---	Step: 15. Fringe size: 24. Depth: 3. Last move: Down. Score: 10 --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 3 | 0 | --- | 1 | 2 | 9 | 0 | ---
Step: 16. Fringe size: 26. Depth: 3. Last move: Left. Score: 10 --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 1 | 2 | 9 | 3 | ---	Step: 17. Fringe size: 28. Depth: 4. Last move: Right. Score: 9 --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 1 | 2 | 3 | 9 | ---	Step: 18. Fringe size: 29. Depth: 2. Last move: Up. Score: 10 --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 9 | 0 | --- | 1 | 2 | 0 | 3 | ---	Step: 19. Fringe size: 32. Depth: 4. Last move: Right. Score: 10 --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 0 | 0 | --- | 0 | 0 | 3 | 0 | --- | 1 | 2 | 0 | 9 | ---

Scalability study

For the scalability, I decided to test how long the algorithms took with blocks removed – that is to say comparing how the program solves the Blocksworld puzzle with just 1 block, A; 2 blocks, A and B; and with 3 blocks, A, B, and C. To do this, I made minor alterations to the parts of the code which checks if the grid is complete and which creates the grid to remove any references to the blocks that did not need to be there. I recorded the number of "steps" necessary to complete the solution which is also the number of nodes checked. This was done five times and an average was taken. Below are the results:



The results are almost shocking in that the random movement of depth first search is surprisingly effective when compared to the other two algorithms which you'd expect to be better. The implications of this result is that the random movement of depth first is superior, at least at this level. One would expect at greater complexities that the heuristic would perform much better than the other methods.

If you are wondering why breadth first search is suspiciously absent, that is because out of memory errors sadly prevented me from exploring deeper than ~14 levels.

Extras and limitations

I am nothing if not humble: there are quite a lot of limitations within my program. For one, the memory issue regarding the breadth-first search essentially makes that method pointless for any reasonably large problem spaces. Additionally, the way depth is calculated is not exactly satisfactory. Its recursive way of iterating up the tree uses a lot of memory and is very slow but I could not find a satisfactory way of doing it better. Finally, my solution is not in the slightest way object oriented with its heavy use of static methods and only one real class.

Code

The two classes are written below:

```
import java.util.*;

public class Main {

  public static void main(String[] args) {
    Node root = new Node();

    //Node leaf = depthFirstSearch(root);
    //Node leaf = breadthFirstSearch(root);
    Node leaf = iterativeDeepening(root);
    //Node leaf = aStarHeuristic(root);


    //Used to obtain the path found by the searches
    //while (leaf.getParent() != null) {
    //    System.out.println(leaf.getLastMove());
    //    leaf = leaf.getParent();
    //}
  }

  public static Node depthFirstSearch(Node root) {
    System.out.println("Starting Depth First Search");
    Stack<Node> fringe = new Stack<Node>(); //LIFO
    fringe.push(root);

    int step = 0;
    //Search loop
    while (true) {

      //Check if fringe size is 0
      if (fringe.size() == 0) {
        System.out.println("Solution not found");
        return null;
      }

      Node node = fringe.pop();
      System.out.println("Step: " + step + ". Fringe size: " + fringe.size() + ". Depth: " + step + ". Last move: " + node.getLastMove() + ". ");
      node.printGridStatus();

      //Check if node is in complete state and solution is found
      if (node.isComplete()) {
        System.out.println("Solution found");
        return node;
      }

      //Add the next level to the fringe
      System.out.println("Adding next level to fringe");
      for (Node nextLevelNode : node.getNextLevel()) {
        if (nextLevelNode != null) fringe.push(nextLevelNode);
      }
      step++;
      System.out.println();
    }
  }

  public static Node breadthFirstSearch(Node root) {
    System.out.println("Starting Breadth First Search");
    Queue<Node> fringe = new LinkedList<Node>(); //FIFO
    fringe.add(root);

    int step = 0;
    //Search loop
    while (true) {
      //Check if fringe size is 0
      if (fringe.size() == 0) {
        System.out.println("Solution not found");
        return null;
```

```
        }

        Node node = fringe.remove();
        System.out.println("Step: " + step /*+ ". Fringe size: " + fringe.size() */+ ". Depth: " + node.getDepth() + ". Last move: " + node.getLastMove() + ". ");
//          node.printGridStatus();

        //Check if node is in complete state and solution is found
        if (node.isComplete()) {
          System.out.println("Solution found");
          return node;
        }

        //Add the next level to the fringe
        System.out.println("Adding next level to fringe");
        for (Node nextLevelNode : node.getNextLevel()) {
          if (nextLevelNode != null) fringe.add(nextLevelNode);
        }
        step++;
        System.out.println();
      }
    }

    static int step = 0;

    public static Node iterativeDeepening(Node root) {
      System.out.println("Starting Iterative Deepening Search");
      for (int depth = 0; depth < 1000; depth++) {
        System.out.println("Current search depth: " + depth);
        Node found = depthLimitedSearch(root, depth);
        if (found != null) {
          System.out.println("Solution found at depth " + found.getDepth());
          found.printGridStatus();
          System.out.println("Number of steps: " + step);
          return found;
        }
      }
      return null;
    }

    private static Node depthLimitedSearch(Node node, int depth) {
      step++;
      if (node.isComplete()) {
        System.out.println("Solution node found!");
        return node;
      }
      if (depth > 0) {
        for (Node child : node.getNextLevel()) {
          Node found = depthLimitedSearch(child, depth - 1);
          if (found != null) return found;
        }
      }
      return null;
    }

    public static Node aStarHeuristic(Node root) {
      System.out.println("Starting A* Heuristic");
      //Priority Queue
      Queue<Node> fringe = new PriorityQueue<Node>(100, (o1, o2) -> o1.getAStarScore() - o2.getAStarScore());
      fringe.add(root);

      int step = 0;

      while(true) {
        //Check if fringe size is 0
        if (fringe.isEmpty()) {
          System.out.println("Solution not found");
          return null;
        }

        Node node = fringe.poll();
        System.out.println("Step: " + step + ". Fringe size: " + fringe.size() + ". Depth: " + node.getDepth() + ". Last move: " + node.getLastMove() + ". Score: " +
node.getAStarScore());
        node.printGridStatus();

        //Check if node is in complete state and solution is found
```

```
            if (node.isComplete()) {
                System.out.println("Solution found");
                return node;
            }

            //Add the next level to the fringe, calculating F value
            for (Node nextLevelNode : node.getNextLevel()) {
                if (nextLevelNode != null) fringe.add(nextLevelNode);
            }

            step++;
        }
    }
}
```

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.PriorityQueue;
import java.util.Queue;

import java.util.ArrayList;
import java.util.Collections;
import java.util.PriorityQueue;
import java.util.Queue;

/**
 * Models the Node of a tree search
 */
public class Node {

    private int[][] grid;
    private int agentY;
    private int agentX;
    private String lastMove;

    private Node parent;
    private int depth;

    //Constructor that sets up the grid to its initial state
    public Node() {
        grid = new int[4][4];
        grid[3][0] = 1; //A
        grid[3][1] = 2; //B
        grid[3][2] = 3; //C

        grid[3][3] = 9; //Agent
        agentX = 3;
        agentY = 3;

        parent = null;
        depth = 0;
    }

    //Constructor for all nodes that aren't the root
    public Node(int[][] grid, int agentY, int agentX, Node parent, String lastMove) {
        this.grid = grid;
        this.agentY = agentY;
        this.agentX = agentX;
        this.parent = parent;
        this.lastMove = lastMove;
        this.depth = 1 + getParent().getDepth();
    }

    //Returns the parent
    public Node getParent() {
        return parent;
    }

    //Returns the depth of the tree
    public int getDepth() {
        if (depth == 0) return 0;
        else return 1 + parent.getDepth();
    }
```

```
//Returns the last move by the node
public String getLastMove() {
   return lastMove;
}

//Returns node with agent moved up or returns null if that is not possible
private Node moveAgentUp() {
   if (agentY == 0) return null;

   int[][] newGrid = grid.clone();
   int swappedBlock = grid[agentY - 1][agentX];

   newGrid[agentY][agentX] = swappedBlock;
   agentY--;
   newGrid[agentY][agentX] = 9;

   return new Node(newGrid, agentY, agentX, this, "Up");
}

//Returns node with agent moved down or returns null if that is not possible
private Node moveAgentDown() {
   if (agentY == 3) return null;

   int[][] newGrid = grid.clone();
   int swappedBlock = grid[agentY + 1][agentX];

   newGrid[agentY][agentX] = swappedBlock;
   agentY++;
   newGrid[agentY][agentX] = 9;

   return new Node(newGrid, agentY, agentX, this, "Down");
}

//Returns node with agent moved left or returns null if that is not possible
private Node moveAgentLeft() {
   //If move is illegal, return null
   if (agentX == 0) return null;

   int swappedBlock = grid[agentY][agentX - 1];
   int[][] newGrid = grid.clone();
   newGrid[agentY][agentX] = swappedBlock;
   agentX--;
   newGrid[agentY][agentX] = 9;

   return new Node(newGrid, agentY, agentX, this, "Left");
}

//Returns node with agent moved right or returns null if that is not possible
private Node moveAgentRight() {
   //If move is illegal, return null
   if (agentX == 3) return null;

   int swappedBlock = grid[agentY][agentX + 1];
   int[][] newGrid = grid.clone();
   newGrid[agentY][agentX] = swappedBlock;
   agentX++;
   newGrid[agentY][agentX] = 9;

   return new Node(newGrid, agentY, agentX, this, "Right");
}

//2D integer array clone method necessary to get the children of the nodes
private int[][] clone(int[][] grid){
   int [][] currentGrid = new int[grid.length][];
   for(int i = 0; i < grid.length; i++)
      currentGrid[i] = grid[i].clone();
   return currentGrid;
}

//Gets the children of the node
public ArrayList<Node> getNextLevel() {
   int [][] currentGrid = clone(grid);
   int agentXStart = agentX;
   int agentYStart = agentY;
```

```
    ArrayList<Node> nodes = new ArrayList<Node>();

    Node moveAgentUpNode = moveAgentUp();
    if (moveAgentUpNode != null && !(moveAgentUpNode.equals(null))) nodes.add(moveAgentUpNode);
    grid = clone(currentGrid);
    agentX = agentXStart;
    agentY = agentYStart;
    Node moveAgentDownNode = moveAgentDown();
    if (moveAgentDownNode != null && !(moveAgentDownNode.equals(null))) nodes.add(moveAgentDownNode);
    grid = clone(currentGrid);
    agentX = agentXStart;
    agentY = agentYStart;
    Node moveAgentLeftNode = moveAgentLeft();
    if (moveAgentLeftNode != null && !(moveAgentLeftNode.equals(null))) nodes.add(moveAgentLeftNode);
    grid = clone(currentGrid);
    agentX = agentXStart;
    agentY = agentYStart;
    Node moveAgentRightNode = moveAgentRight();
    if (moveAgentRightNode != null && !(moveAgentRightNode.equals(null))) nodes.add(moveAgentRightNode);
    grid = clone(currentGrid);
    agentX = agentXStart;
    agentY = agentYStart;

    Collections.shuffle(nodes);

    return nodes;
}

//Returns true if the grid is in its completed state
public boolean isComplete() {
    if (    grid[1][1] == 1 && //A
            grid[2][1] == 2 && //B
            grid[3][1] == 3 && //C
            grid[3][3] == 9) { //Agent
        return true;
    }
    else return false;
}

//Prints the status of the grid
public void printGridStatus() {
    System.out.println("-----------------");
    for (int[] gridLines : grid) {
        String line = "| ";
        for (int j = 0; j < grid.length; j++) {
            line = line + gridLines[j] + " | ";
        }
        System.out.println(line);
        System.out.println("-----------------");
    }
}

public int getAStarScore() {
    int g = getDepth();
    int h;

    int aX = 0;
    int aY = 0;
    int bX = 0;
    int bY = 0;
    int cX = 0;
    int cY = 0;

    //Find a, b, c x values and y values
    for (int y = 0; y < grid.length; y++) {
        for (int x = 0; x < grid.length; x++) {
            if (grid[y][x] == 1) {
                aX = x;
                aY = y;
            }
            if (grid[y][x] == 2) {
                bX = x;
                bY = y;
            }
            if (grid[y][x] == 3) {
```

```
                cX = x;
                cY = y;
              }
            }
          }

          //Manhattan distance from a, b, c, agent to goal state
          h =   Math.abs(aX - 1) + Math.abs(aY - 1)
             + Math.abs(bX - 1) + Math.abs(bY - 2)
             + Math.abs(cX - 1) + Math.abs(cY - 3)
             + Math.abs(agentX - 3) + Math.abs(agentY - 3);

          return g + h;
        }
}
```