Blind and Heuristic Search Methods and Algorithms

COMP2208 Intelligent Systems
Assignment Report

Roberto J Gregoratti
Student ID: 25270176
rjg1g12@ecs.soton.ac.uk

January 9th, 2014

University of Southampton
Faculty of Physical Sciences and Engineering
School of Electronics and Computer Science

SECTION I - The assignment

1.1 - Objective of the assignment

For this assignment we had to design an intelligent application to move an agent around a ‘world’ (grid) with blocks with the goal of building towers (the final states of the ‘world’). Some blocks would have a name (the ones we had to stack, i.e. A, B and C) and one would be the agent (represented as we wanted to). The specification asked us to make the agent be able to move up, down, left or right, with the agent swapping positions with the tile in whose direction it had moved. Furthermore, to achieve the goal states we were asked to implement both uninformed searches (breadth-first, depth-first and iterative deepening) and one type of heuristic search.

The purpose of this report is to present evidence of the specifications having been followed and to show that the methods have been correctly implemented and work as expected, and furthermore to present the approach taken and explain how the application has been designed, tested and works.

1.2 - Description of the approach taken

For the purpose of this assignment I chose the Java programming language, as it offers a variety of data structures that would have helped me to write efficient code for this application.
My personal interpretation of the assignment was to have as start and end ‘worlds’ the ones given in the coursework specification (and similar, for bigger board sizes), so in my implementation the start and end nodes are fixed. An example of this setup can be seen in the illustration below:

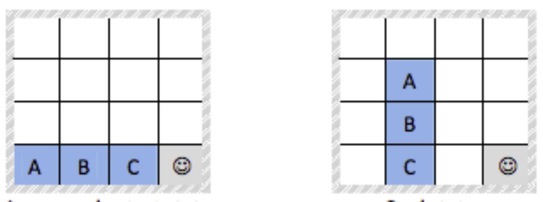


Figure 1 – 4x4 Start and End states (from spec)

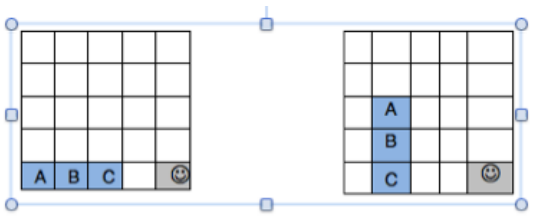


Figure 2 - 5x5 Start and End states (my board)

As can be seen by the examples provided above, my implementation sees the start states having the three tiles to stack positioned at the start in the bottom left corner of the grid, while the end states have the three tiles positioned at the bottom three places in the second column from the left. In both the start and goal states, the agent is in the bottom right corner of the grid. Positions could, of course, be changed by modifying some of the source code (provided in Appendix A of this report), therefore also changing the solution depths obtained but not altering the time complexities and algorithms’ performances.

My design of the grid world for this assignment is based on *Block* instances that live in a *BlocksWorld* (a class containing the world’s start and end nodes, and methods to perform the searches) which has, at any point, one *State* of the grid, stored in the *Node* corresponding to it. A *TestHarness* class used originally to perform all the tests to check that the ‘infrastructure’ worked correctly was created, then once the tests were successful (position checking, moving the agent, etc.) a method called *createSizedWorld(int i)* was added, that takes an int for the size of the world and creates a corresponding grid, as well as the tiles positioned as explained earlier, creating the start and end *State*s. Once the world is created, *performSearches()* is called, which calls the methods to execute the various searches on the current world, saving the output to a file for analysis.

A *Block* has a name (String, being “A”/”B”/”C”, “X” for the Agent and “Empty” for all other tiles), and an x/y coordinate system to indicate its position in the grid (2D array of *Block* objects), with methods to return its position.

A *State* represents a board configuration. A toString() method returning it as a String containing the positions of the 4 ‘special’ tiles only has been added, and methods to get and set the position of a tile, compare it to another *State* (two methods, one to compare it to the goal state with the agent being fixed and one with the agent allowed to end anywhere) and a further method to move the agent in the board in the cardinal directions (if allowed).

A *Node* is the object explored in the searches. It represents the state of the world at any time, contains the *State* of the board, its parent node and information like the heuristic cost estimate to the final state of the world. It has accessor and mutator methods and a method to return an Array of all *Node*s that are neighbours (which can be explored from it), obtained by trying to make the agent of the current board state move in all directions.

Before creating the class containing the search methods, a test harness was used to test the correct positioning of the blocks and the correct configuration of the board when moving the agent. Some example test world were hard-coded and tests were performed on them. Once the testing results matched those expected, the search methods were created. The full testing code is included in the *TestHarness* class provided in Appendix A of this report.

SECTION II - Results’ and scalability investigation

2.1 – Design and implementation of the search methods

2.1.1 – Breadth-First Search (BFS)

My design of the BFS algorithm was based on the pseudocode obtained from the references presented in section 3 of this report. It uses a Queue data structure to contain the nodes to analyse and a *Set* (*HashSet*) for the nodes that have already been visited. At the start the initial node is added to both structures, then, as long as the queue is not empty, the first node is obtained by using the *poll()* method. If the state obtained matches the goal state, the search ends, prints the results to the output for the final node and returns it. If a match is not found, an *ArrayList* of *Node*s representing the possible moves of the agent to the neighbours is created, and for each possible one there is a check to see if it has already been visited. If that is the case it is ignored, else it’s added to both data structures, for analysis. The method will keep running until a state matching the goal is found. If none are found an error occurred, so null is returned and an error message is printed.

2.1.2 – Depth-First Search (BFS)

My design of the DFS algorithm is similar to the one presented in 2.1.1 for BFS, with the only change being the fact that a Stack is used rather than a queue to store the nodes that still have to be visited, and therefore the operations used to retrieve the nodes and add the nodes to the stack are *pop()* and *push()*, respectively.

2.1.3 – Iterative Deepening (Depth-First) Search (IDS)

Adapting DFS to search only up to a maximum depth, iteratively increasing, has created the IDS algorithm. A helper method was created, taking start and goal nodes (and other parameters used exclusively to print the results obtained at the end when the result is found). At the beginning a null node and an int for the depth, initially 1, are created. While the node is null (and no end node is found and passed back), the null node is set to be equal to the node (if any) returned by calling the *performIDS()* method to perform the DFS, using the start and finish nodes and the depth, increased after each call. When a match is found the solution is printed to the output, and the node returned. The *performIDS()* method works like the DFS one, except for the fact that the HashSet has been substituted with a *HashMap<Node,Integer>* , to store the depth at which each node has been visited. Everything else works just as the DFS algorithm from 2.1.2, with the exception of depth checking when analyzing nodes. When a node matching the goal is finally found, the helper method is called one final time, passing back the end node and end time too, allowing the helper method to print the result’s data to the output.

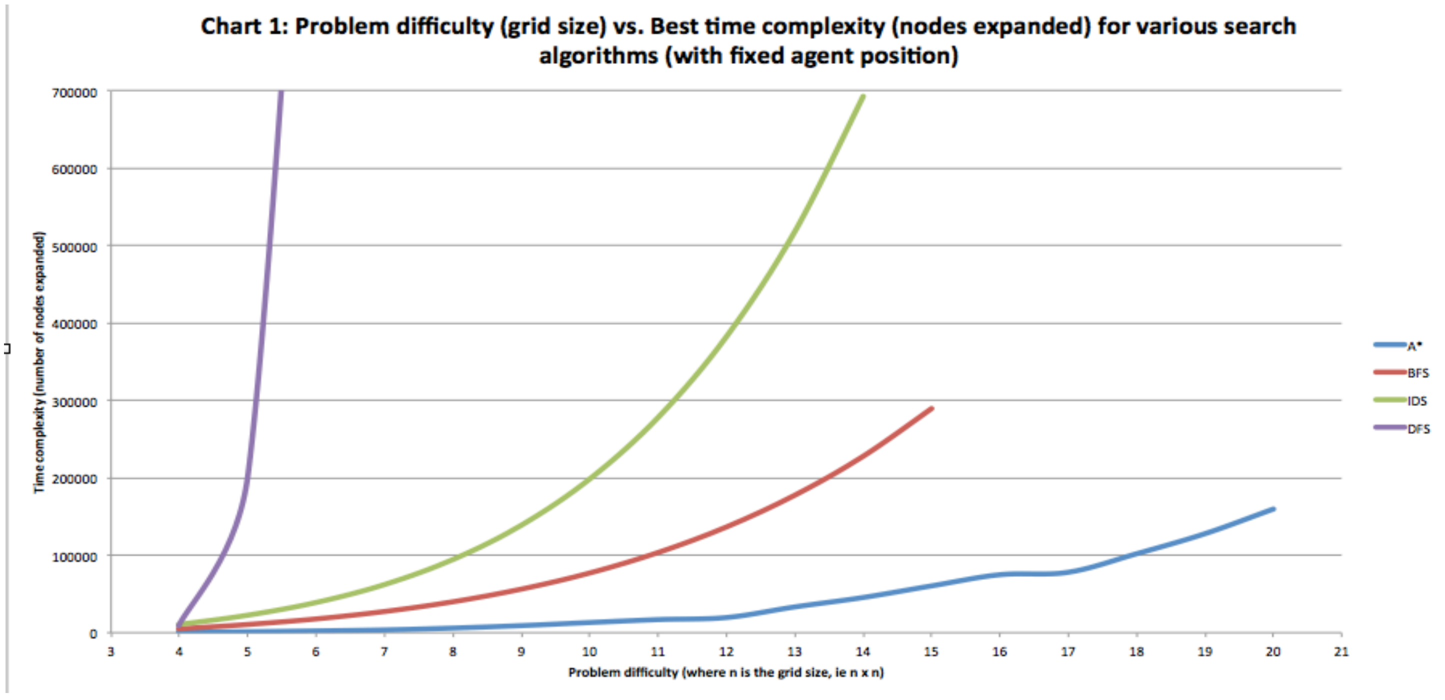
2.1.4 – A* (Heuristic) Search

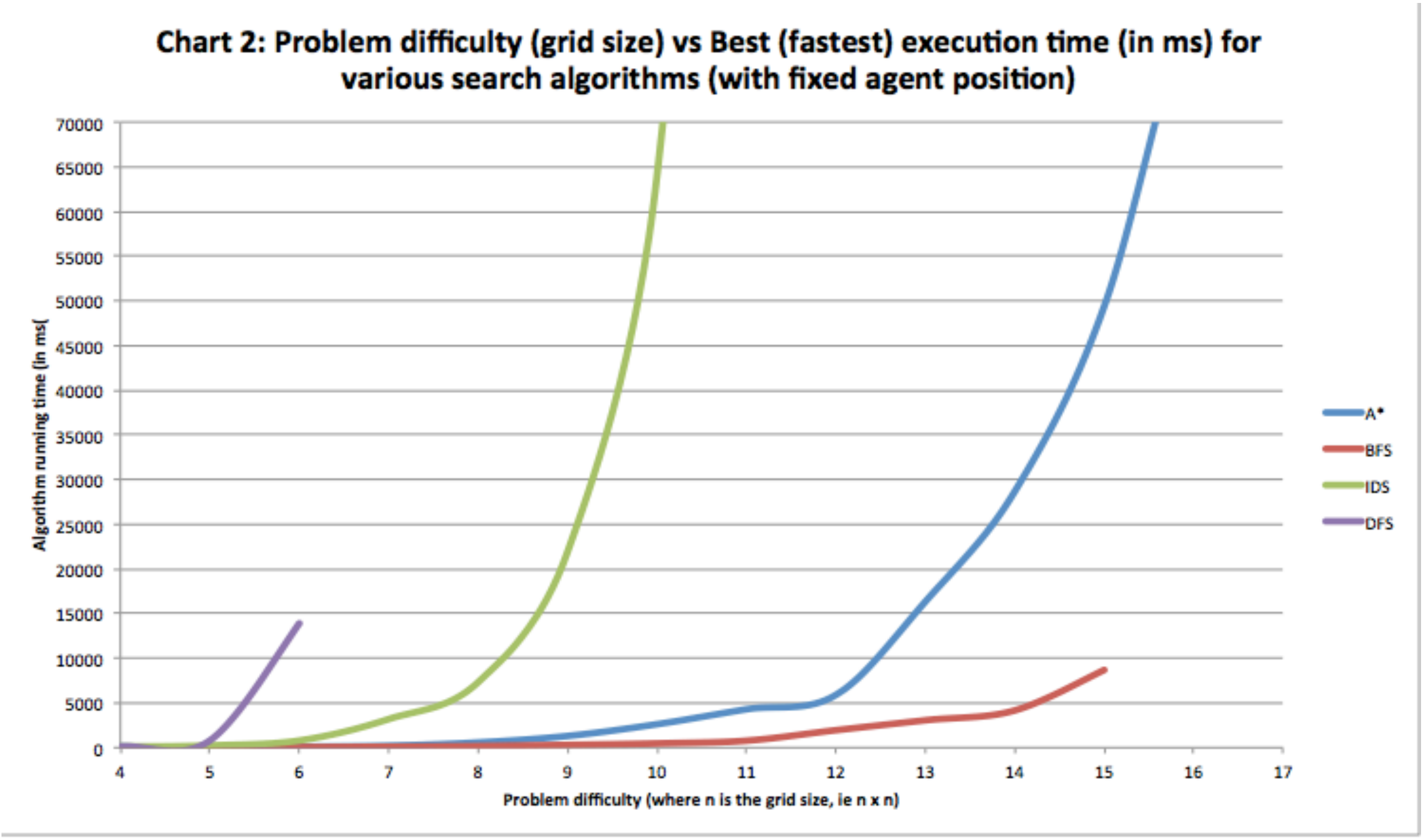
A* has also been adapted from pseudocode from different sources. It works similarly to BFS algorithm, using a queue and set, with the only difference being that it uses heuristics to estimate the cost from a node to the finish node. At the beginning, this is done for the start, which is then added to the queue. While there are still nodes to analyse, we *poll()* the queue to get the first one, compare it to the finish state, return its result and the node to complete the search if it matches the output. If not, add it to the visited nodes, get all possible moves (neighbours), then for each neighbour if it isn't in the set and queue, find the estimated cost to the finish node and add it to the queue, continuing until the node matching the given end node is found.

2.2 - Data collection and analysis of results

Once the methods had been created and tested on boards of size 4x4 (see section 2.5 for the testing details and screenshots), a *PrintStream* was created and added to the main class to send the data to a CSV file for analysis. To allow for scalability and performance analysis, the worlds' problem difficulties (ie, grid sizes) were increased, and data was obtained for boards of size 4x4 to 20x20. The obtained data was then analysed, and graphs were created to compare the time complexities of the various algorithms (ie, number of nodes explored vs. grid size) and the real times needed to run the algorithms (ie, execution time in milliseconds).

Presented below are the graphs obtained:





From the data presented in these graphs we can conclude the following:

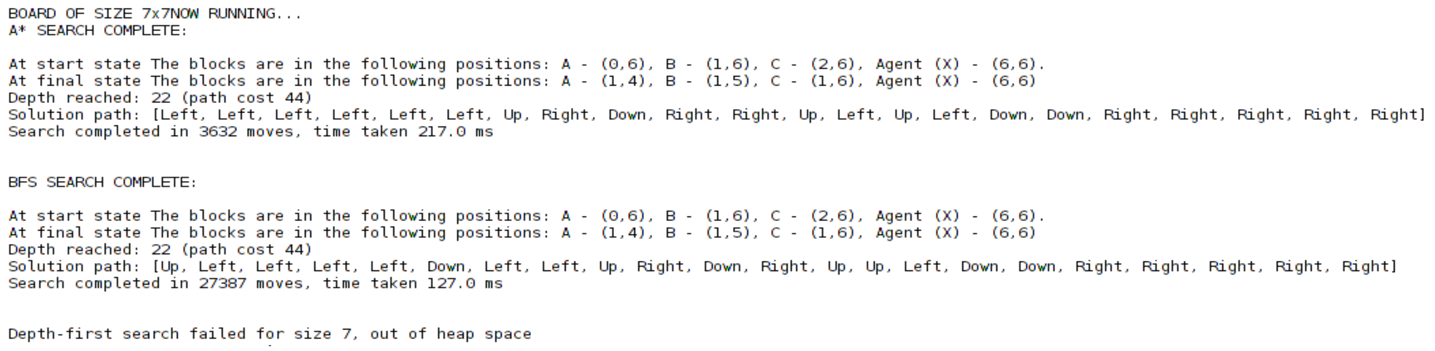
- **Depth-First Search** (DFS) presents the user with a great imbalance between speed and efficiency, exploring a huge number of nodes in a fast (enough) time. It fails after problem sizes of around 6x6 or 7x7 in my implementation, demonstrating its suitability to solve only small problems. However, the solution it finds isn't optimal, so it **should be the last resort** as a search algorithm as it uses a lot of resources and is terribly inefficient on both runtime and time complexity;
- **Breadth-First Search** (BFS) is a good algorithm, the 2nd best for time/space complexity and the best for execution time (NOTE: a limitation of these findings is that the way my algorithms are coded, the machines these tests have been executed on and my heuristic estimates might have limited the speed of A* therefore making BFS be faster, in my implementation exclusively!). It presents the user with a great trade-off between speed and efficiency, however it fails around 15x15-20x20 in my implementation. It is, however, optimal;
- **Iterative Deepening DFS** (IDS) is better than DFS and is also optimal, but however still inefficient, expanding huge numbers of nodes in a lot of time (though less than DFS, for both). This algorithm could be used, if necessary, but failed around 12x12-15x15 depending on the machine it was ran on);
- Finally, **A* Search** (heuristic) was the overall best. It presents the user with great efficiency (time complexity) and great search time (which could, however, be further improved by using better heuristic estimates). It is also optimal and **is the algorithm to be preferred**, as was expected, for a search problem of this kind.

2.3 - Scalability investigation

To ensure that the algorithms worked on problems of different sizes, all tests were ran on boards of sizes 4 to 20. States for 4x4 and 5x5 sizes were originally hard-coded, but for practicality the *TestHarness* class was later adapted to create a board of the next size once the test on the previous one was finished: *createSizedWorld(int i)* automatically calls the BlocksWorld constructor passing it parameters obtained from manipulating the desired grid size. Once a board of bigger size had been created, the testing methods would be called on it, automatically, and after they finished running the program would proceed to analyse a larger state. As a result of this, data for problem sizes of 4 to 20 was obtained, and included in the graphs presented in section 2.2. Scalability testing was successful and the algorithms worked on any problem size (up to the points where they failed). From the investigations run on different machines it has been noticed that my search algorithms will fail at different problem sizes varying for every machine (for example, IDS failed at 12x12 or 15x15). For the purposes of this report the data corresponds to the algorithms' best performances (ie, the highest problem size before they failed). Scalability in terms of tile positions has also to be considered: my interpretation of this assignment has a fixed start and end state for every problem size, but the user might want problems with different start and end positions. However, the data plotted in the graphs presented in Section 2.2 should be consistent, and the algorithms' performances the same, regardless of the start and end position of the tiles (as long as the agent is fixed).

2.4 - Testing and evidence

Once the methods were created, tests were carried out on a board of size 4 to ensure the methods worked as expected. By previous analysis it had been identified that the optimal depth at which the methods could find solution for a grid of size 4, given the start and end states provided in section 1.2 of this report, was 16. As expected, when accounting for the agent's final position, the depth at which the optima algorithms BFS, IDS and A* found a solution was 16 (ie, the solution could be achieved in 16 moves). Similarly, tests were carried out on boards of sizes up to 20x20, the solution paths and optimal depths analysed and identified to be correct.
Below are some screenshots demonstrating the algorithms working with an example failure of DFS at 7x7 and solutions found up to size 11x11 (4x4 full results can be seen in Appendix B of this report):



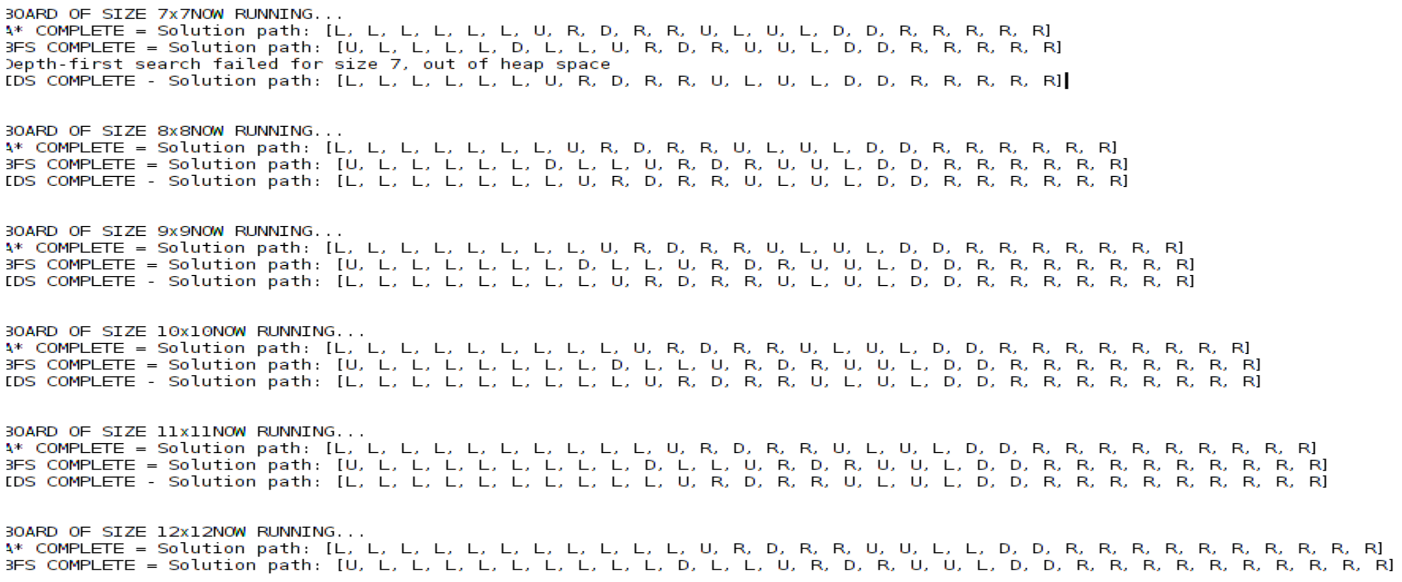
```
BOARD OF SIZE 7x7NOW RUNNING...
A* SEARCH COMPLETE:

At start state The blocks are in the following positions: A - (0,6), B - (1,6), C - (2,6), Agent (X) - (6,6).
At final state The blocks are in the following positions: A - (1,4), B - (1,5), C - (1,6), Agent (X) - (6,6)
Depth reached: 22 (path cost 44)
Solution path: [Left, Left, Left, Left, Left, Left, Up, Right, Down, Right, Right, Up, Left, Up, Left, Down, Down, Right, Right, Right, Right, Right]
Search completed in 3632 moves, time taken 217.0 ms


BFS SEARCH COMPLETE:

At start state The blocks are in the following positions: A - (0,6), B - (1,6), C - (2,6), Agent (X) - (6,6).
At final state The blocks are in the following positions: A - (1,4), B - (1,5), C - (1,6), Agent (X) - (6,6)
Depth reached: 22 (path cost 44)
Solution path: [Up, Left, Left, Left, Left, Down, Left, Left, Up, Right, Down, Right, Up, Up, Left, Down, Down, Right, Right, Right, Right, Right]
Search completed in 27387 moves, time taken 127.0 ms


Depth-first search failed for size 7, out of heap space
```



```
3OARD OF SIZE 7x7NOW RUNNING...
\* COMPLETE = Solution path: [L, L, L, L, L, L, U, R, D, R, R, U, L, U, L, D, D, R, R, R, R, R]
3FS COMPLETE = Solution path: [U, L, L, L, L, D, L, L, U, R, D, R, U, U, L, D, D, R, R, R, R, R]
)epth-first search failed for size 7, out of heap space
[DS COMPLETE - Solution path: [L, L, L, L, L, L, U, R, D, R, R, U, L, U, L, D, D, R, R, R, R, R]


3OARD OF SIZE 8x8NOW RUNNING...
\* COMPLETE = Solution path: [L, L, L, L, L, L, L, U, R, D, R, R, U, L, U, L, D, D, R, R, R, R, R, R]
3FS COMPLETE = Solution path: [U, L, L, L, L, L, D, L, L, U, R, D, R, U, U, L, D, D, R, R, R, R, R, R]
[DS COMPLETE - Solution path: [L, L, L, L, L, L, L, U, R, D, R, R, U, L, U, L, D, D, R, R, R, R, R, R]


3OARD OF SIZE 9x9NOW RUNNING...
\* COMPLETE = Solution path: [L, L, L, L, L, L, L, L, U, R, D, R, R, U, L, U, L, D, D, R, R, R, R, R, R, R]
3FS COMPLETE = Solution path: [U, L, L, L, L, L, L, D, L, L, U, R, D, R, U, U, L, D, D, R, R, R, R, R, R, R]
[DS COMPLETE - Solution path: [L, L, L, L, L, L, L, L, U, R, D, R, R, U, L, U, L, D, D, R, R, R, R, R, R, R]


3OARD OF SIZE 10x10NOW RUNNING...
\* COMPLETE = Solution path: [L, L, L, L, L, L, L, L, L, U, R, D, R, R, U, L, U, L, D, D, R, R, R, R, R, R, R, R]
3FS COMPLETE = Solution path: [U, L, L, L, L, L, L, L, D, L, L, U, R, D, R, U, U, L, D, D, R, R, R, R, R, R, R, R]
[DS COMPLETE - Solution path: [L, L, L, L, L, L, L, L, L, U, R, D, R, R, U, L, U, L, D, D, R, R, R, R, R, R, R, R]


3OARD OF SIZE 11x11NOW RUNNING...
\* COMPLETE = Solution path: [L, L, L, L, L, L, L, L, L, L, U, R, D, R, R, U, L, U, L, D, D, R, R, R, R, R, R, R, R, R]
3FS COMPLETE = Solution path: [U, L, L, L, L, L, L, L, L, D, L, L, U, R, D, R, U, U, L, D, D, R, R, R, R, R, R, R, R, R]
[DS COMPLETE - Solution path: [L, L, L, L, L, L, L, L, L, L, U, R, D, R, R, U, L, U, L, D, D, R, R, R, R, R, R, R, R, R]


3OARD OF SIZE 12x12NOW RUNNING...
\* COMPLETE = Solution path: [L, L, L, L, L, L, L, L, L, L, L, U, R, D, R, R, U, U, L, L, D, D, R, R, R, R, R, R, R, R, R, R]
3FS COMPLETE = Solution path: [U, L, L, L, L, L, L, L, L, L, D, L, L, U, R, D, R, U, U, L, D, D, R, R, R, R, R, R, R, R, R, R]
```

SECTION III - Conclusion and Critical Evaluation

3.1 – Conclusions and significance

To sum up the findings presented within this report, the reader can see the comparison between the various algorithms in the charts presented in Section 2.2 of this report. From the analyses of the data obtained with the algorithms and tests presented in this report, it emerged that the A* heuristic search algorithm is the one to be preferred, offering a great balance between time complexity and runtime. Where A* can't be used, then of the uninformed search algorithms BFS is to be preferred, as it was the 2nd best in the tests. However, a great limitation that has to be considered for all algorithms presented in this report is the way they have been coded: little tweaking might greatly improve the algorithms and their performances (see 3.3 for further limitations).

3.2 – Assumptions and personal interpretation

What also has to be considered when analyzing the results presented in this report is my personal interpretation of the assignment. Given the 'fixed' nature of the start and end states, to allow for other states it would be necessary to change some of the code written. Some tests have also been run (but aren't included in this report) with the agent being allowed to end anywhere, but the values returned by A* are inconsistent due to the inadequacy of the chosen heuristic evaluation if the agent doesn't end in a fixed position, so a change is needed.

3.3 – Limitations and weaknesses of this work

First of all, there are limitations regarding the algorithms: it is possible to obtain better performances (though not better optimal solutions) by coding more efficient algorithms that might work on a larger space state than the one they fail at in the examples presented. Another limitation the reader needs to pay attention to regards the heuristic estimates, as there might be one that would make the algorithm perform better than what has been presented. Some further limitations to consider are, for example, the small number of tests that have been carried out, the potential to rewrite some parts of the code that might be inefficient or to optimize the existing code, the space complexity being affected by the usage of certain data structures like HashSet and HashMap to detect loops and visited nodes, and more. A final factor identified to be a potential limitation for the accuracy of the data is the choice of programming language. Choosing a simpler, faster programming language, although it might limit the choice of data structures and possibilities when writing the code, might result in better performance and faster algorithms. Given the problems that arose during the work for this assignment caused by the exhaustion of heap space in Java, not only might writing better algorithms or using a different programming language improve the accuracy of the data, but also it might solve some problems related to the memory usage of the algorithms (or cause more, if not paid attention to).

Further improvements could be made by building a graphical user interface to visualize the board state at every move of the agent, or by restructuring the code to have less method calls and/or more efficient move checking (e.g. by using different data structures).

In conclusion, the data presented in this report might be inaccurate, but it could be improved by simply altering the algorithms to enhance performance, coming up with better heuristics and carrying out more extensive testing on many more cases than the ones analysed for the purposes of this assignment (for example, by allowing the agent to end anywhere or using different start and end problems).

3.X – References

"A* Search", at http://www.peachpit.com/articles/article.aspx?p=101142&seqNum=2
Guidance from lecture slides and course material
Russell & Norvig, *Artificial Intelligence: A Modern Approach (2nd ed.),* Prentice Hall
Wikipedia articles on the search methods

APPENDIX A
- Source code for the application -

TestHarness class

```java
import java.io.File;
import java.io.PrintStream;

/*
This class is a test harness written to firstly test that the 'infrastructure' for the blocksworld
worked, then to create
and initialise different worlds to do their searches. There are some booleans to store the feasibility
of the various
searches which will be toggled to change to false when those searches run out of memory space, and
methods to create
different worlds (automated) and initialise them to perform their searches.
 */

public class TestHarness {

    static boolean dfsOK = true;        //booleans representing search feasibility
    static boolean bfsOK = true;
    static boolean idsOK = true;
    static boolean heurOK = true;

    public static void main(String[] args) throws Exception{
        try{             //test to see if states created properly (error checking done previously)
            //State state = new State(1,1,1,2,1,3,2,2);
            //System.out.println(state.toString());
            //State state2 = new State(1,1,1,2,1,3,2,1);
            //State state3 = new State(1,1,1,2,1,3,2,2);

            /*
            Preliminary tests

            System.out.println(state.compareTo(state2));    //comparison and goal achievement checking
            System.out.println(state.compareTo(state3));
            System.out.println(state.equalToGoal(state2));

            state.moveAgent(Move.UP);       //brings test agent to 2,1
            System.out.println("\n================\n");

            state.moveAgent(Move.DOWN);     //brings test agent to 2,2
            System.out.println("\n================\n");

            state.moveAgent(Move.DOWN);     //brings test agent to 2,3
            System.out.println("\n================\n");

            state.moveAgent(Move.UP);       //brings test agent to 2,2
            System.out.println("\n================\n");

            state.moveAgent(Move.LEFT);       //brings test agent to 1,2, B to 2.2
            System.out.println("\n================\n");

            state.moveAgent(Move.RIGHT);       //brings test agent to 2,2, B to 1,2
            System.out.println("\n================\n");

            */

             /*
            Move check testing
            System.out.println("Testing moves:");
            for(Node node : world1.current.checkMoves()){
               System.out.println(node.getState());
            }

            */

            //stream to print results to file
            PrintStream out = new PrintStream(new File("SearchResultsAgent.csv"));

            //create worlds (grids) to perform tests/searches on, sizes 4x4 to 20x20
              for(int i=4;i<=20;i++){
                createSizedWorld(i,out);
            }

        }
        catch(Exception e){     //catch any exceptions that may arise
            e.printStackTrace();
        }

    }

/*
    This method creates a world of the size provided. In my interpretation of the coursework specs and
implementation
    of the blocksworld, the tiles have fixed positions (and are therefore created automatically with
the size:
    in the start state the tiles will be in the bottom row, first 3 tiles from the left, with the
agent being in the
    bottom right corner. The end state will always be with the agent in the bottom right corner and
the stacked tiles
    in the bottom three tiles of the second column.
 */
    public static void createSizedWorld(int size,PrintStream out) throws Exception{
        //Create the world (grid) of the specified tiles, with the tiles' positions fixed for start
and end
        int minusOne = size - 1;        //helper variables for board state creation
        int minusTwo = size - 2;
        int minusThree = size - 3;
        BlocksWorld temp = new BlocksWorld(0,minusOne,1,minusOne,2,minusOne, minusOne,
minusOne,1,minusThree,1,minusTwo,1,minusOne,minusOne,minusOne,size,out);
        //created using set parameters (read method documentation, above)

        try{
            performSearches(temp,size);
        }
        catch(Exception e){
            e.printStackTrace();
        }

    }

    /*
    Method to perform the various searches on the given world and output the results. For every
search, check if
    it's still feasible (ie, if boolean hasn't been toggled to false at the earlier iterations), then
if it is
    perform the relative search, catch OOM Exception and return message otherwise, toggling the
feasibility boolean
    to be set to false (not feasible anymore)
     */
    public static void performSearches(BlocksWorld world, int size) throws Exception{

        System.out.println("\n\nBOARD OF SIZE " + size + "x" + size + "NOW RUNNING...");

        if(heurOK){
            try{
                world.heuristic(world.current, world.finish);
            }
            catch(OutOfMemoryError oome){
                System.out.println("Heuristic search failed for size " + size + ", out of heap
space");
                heurOK = false;
            }
            catch(Exception e){
                e.printStackTrace();
            }
        }

        if(bfsOK){
            try{
                world.breadthFirst(world.current, world.finish);
            }
            catch(OutOfMemoryError oome){
                System.out.println("Breadth-first search failed for size " + size + ", out of heap
space");
                bfsOK = false;
            }
            catch(Exception e){
                e.printStackTrace();
            }
        }

        if(dfsOK){
            try{
                world.depthFirst(world.current, world.finish);
            }
            catch(OutOfMemoryError oome){
                System.out.println("Depth-first search failed for size " + size + ", out of heap
space");
                dfsOK = false;
            }
            catch(Exception e){
                e.printStackTrace();
            }
        }

        if(idsOK){
            try{
                world.iterativeDeepening(world.current, world.finish, null, 0);
            }
            catch(OutOfMemoryError oome){
                System.out.println("Iterative deepening search failed for size " + size + ", out of
heap space");
                idsOK = false;
            }
            catch(Exception e){
                e.printStackTrace();
            }
        }
    }
}
```

BlocksWorld class

```
import java.io.PrintStream;
import java.util.*;

/*
This class represents an instance of the assignment's problem, the blocksworld. It has a specific set-
up
of tiles (provided by the main method through various tile-placing strategies), start and finish
nodes,
a PrintStream it receives by the main class when created to print the results to a CSv file for
analysis
purposes, and methods to perform the different kinds of heuristic and uninformed searches.
The uninformed searches implemented are: BFS, DFS, ID(DF)S.
The heuristic search I chose to implement is: A*.
 */

public class BlocksWorld{

    PrintStream out;        //obtained from main class, to print data to a file for analysis

    Node finish;                    //end node
    Node current;                   //start state
    int totalIDSMoves;          //variable recording the total number of moves performed by IDS during
all iterations

    /*
    The constructor takes int parameters to specify the position of the tiles on the grid for both the
start and goal states,
    the size of the grid (worldSize) and the printstream coming from the main classes. The start and
finish nodes are
    initialised with those parameters.
     */

    public BlocksWorld(int xa, int ya, int xb, int yb, int xc, int yc, int xAg, int yAg, int finXA,
int finYA, int finXB, int finYB, int finXC, int finYC, int finXAg, int finYAg, int worldSize,
PrintStream out) throws Exception {
        finish = new Node(new State(finXA,finYA,finXB,finYB,finXC,finYC,finXAg,finYAg,worldSize));
        current = new Node(new State(xa,ya,xb,yb,xc,yc,xAg,yAg, worldSize));
        this.out = out;
    }

    /*
    Method to perform breadth-first search (BFS), adapted from pseudocodes found on Wikipedia, lecture
slides and
    AIAMA2e (Russell-Norvig)
     */

    public Node breadthFirst(Node start, Node finish) throws Exception{

        long start_time = System.currentTimeMillis();      //real-time registered (at the beginning)
        Queue queue = new LinkedList();          //DS containing nodes to analyse
        HashSet set = new HashSet();            //DS containing nodes visited (to avoid repetitions)
        queue.add(start);           //start node added to both queue and set (at start)
        set.add(start);

        while(!queue.isEmpty()){                //while there are still nodes left to analyse
            Node node = (Node) queue.poll();        //get first node to analyse from the queue

            //when element in queue matches, terminate search and print results/get total search time
            if(node.getState().compareTo(finish.getState())){
                double time_end = (System.currentTimeMillis() - start_time);
                printResults("BFS",start,node,time_end,set);
                return node;
            }

            ArrayList<Node> possibleMoves = node.checkMoves();      //get possible moves of the

            //if neighbour not visited yet, add to both queue and set
            for(Node n : possibleMoves){
                if (!set.contains(n)){
                    set.add(n);
                    queue.add(n);
                }
            }
        }
        System.out.println("Error occurred while running BFS! Search failed!");
        return null;              //nothing found, null returned
    }


    /*
    Method to perform depth-first search (DFS), adapted from pseudocodes found on Wikipedia, lecture
slides and
    AIAMA2e (Russell-Norvig). Works similarly to BFS one above, minor modifications, uses stack
instead of queue DS
     */

    public Node depthFirst(Node start, Node finish) throws Exception{
        long start_time = System.currentTimeMillis();

        Stack stack = new Stack();        //use stack DS to store nodes to visit, not queue (like BFS)
        HashSet set = new HashSet();
        stack.push(start);
        set.add(start);

        while(!stack.isEmpty()){
            Node node = (Node) stack.pop();     //pop element to analyse from stack

            //search completed when what's popped from the stack matches goal, search completed.
            if(node.getState().compareTo(finish.getState())){
                double time_end = (System.currentTimeMillis() - start_time);
                printResults("DFS",start,node,time_end,set);
                return node;
            }

            ArrayList<Node> possibleMoves = node.checkMoves();

            for(Node n : possibleMoves){
                if (!set.contains(n)){
                    set.add(n);
                    stack.push(n);
                }
            }
        }

        System.out.println("Error occurred while running DFS! Search failed!");
        return null;       //nothing found - null returned
    }

    /*/
    Two methods to perform Iterative Deepening Search (IDS). The first method starts from depth 1,
creates null
    node, calls the IDS-performing method and stores the returned Node in n. While still null,
continues looping
    to find IDS solution. Depth is increased at every iteration.
    IDS code in performIDS() is adapted from pseudocode found on Russell-Norvig, websites and lecture
slides
    NOTE: the 'end' and 'time' variables passed to the iterativeDeepening() method are just for
result-printing
    purposes due to the fact that the search is split between two methods!
     */

    public Node iterativeDeepening(Node start, Node finish,Node end,double time) throws Exception{

        long start_time = System.currentTimeMillis();

        int depth = 1;      //initial depth
        Node n = null;      //null node created, will be equal to what the DFS returns at 'depth'

        if(end == null){
            while(n == null){
                n = performIDS(start, finish, depth,start_time);        //perform DFS with max depth
                depth++;            //increase depth, after having set n to be what the IDDFS returned
            }
        }
        else{
            //when match found, print to output the results
            out.println("IDS" + "," + start.getState().gridDimension + "," + end.getDepth() + "," +
totalIDSMoves + "," + time);
        }

        return n;
    }


    public Node performIDS(Node start, Node finish, int depth, long time) throws Exception{     //DFS
with limited depth (iterative deepening)

        Stack stack = new Stack();      //uses stack DS like DFS
        Map<Node,Integer> map = new HashMap<Node, Integer>();   //Map DS to store visited nodes/depths
        stack.push(start);
        map.put(start, start.getDepth());       //start node entered in Map with initial depth

        while(!stack.isEmpty()){
            Node node = (Node) stack.pop();

            if(node.getState().compareTo(finish.getState())){
                double end_time = (System.currentTimeMillis() - time);
                iterativeDeepening(start,finish,node,end_time);
                return node;
            }

   //if the node at smaller depth than target maximum depth for the search, check if in map, etc.
            if(depth > node.getDepth()){
                ArrayList<Node> possibleMoves = node.checkMoves();
                for(Node n : possibleMoves){
                    if (!map.containsKey(n) || map.get(n) >= n.getDepth()){
                        map.put(n, n.getDepth());
                        stack.push(n);
                    }
                }
            }
        }

        totalIDSMoves += map.size(); //add the number of visited nodes in this iteration to the global
        return null;
    }


    /*
    Method to perform an A* heuristic search, adapted from pseudocode in Russell-Norvig, on Wikipedia
and online.
     */

    public Node heuristic(Node start, Node finish) throws Exception{
        long start_time = System.currentTimeMillis();

        Queue queue = new PriorityQueue();
        HashSet set = new HashSet();
        start.getCostEstimate(finish);      //create cost estimate for path to finish node
        queue.add(start);

        while(!queue.isEmpty()){
            Node current = (Node) queue.poll();

            if(current.getState().compareTo(finish.getState())){
                double time_end = (System.currentTimeMillis() - start_time);
                printResults("A*",start,current,time_end,set);
                return current;
            }

            set.add(current);
            ArrayList<Node> possibleMoves = current.checkMoves();

            for(Node n : possibleMoves){
                if (!set.contains(n) && !queue.contains(n)){
                    n.getCostEstimate(finish);   //create and store estimate from neighbour to finish
                    queue.add(n);
                }
            }
        }
        return null;
    }

    /*
    Method used to test that the methods worked. Printed to output console various parameters, like
start and end configurations of the
    world board, depth of solution, moves and time taken to reach it, path cost. Commented out due to
the newer one printing to csv files.

    public void printResults(String search, Node start, Node current, double srcTime, Set set){
        System.out.println(search + " SEARCH COMPLETE:\n\nAt start state " +
start.getState().toString() +".\nAt final state " + current.getState().toString()
            + "\nDepth reached: " + current.getDepth() + " (path cost " + current.getCost() + ")");
        System.out.println("Solution path: " + Arrays.toString(current.displaySolution()));
        System.out.println("Search completed in " + set.size() + " moves, time taken " + srcTime + "
ms\n\n");
    }
    */

    /*
    Method used to test the solution path printing.

    public void printResults(String search, Node start, Node current, double srcTime, Set set){
        System.out.println(search + " COMPLETE = Solution path: " +
Arrays.toString(current.displaySolution()));
    }
    */

    /*
    Method to print the searches' results. Takes and prints in CSV format some data about search
results on various search spaces,
    prints it to the file the output stream is linked to.
     */

    public void printResults(String search, Node start, Node current, double srcTime, Set set) throws
Exception{
        out.println(search + "," + current.getState().gridDimension + "," + current.getDepth() + "," +
set.size() + "," + srcTime);
    }
}
```

Block class

```
/*
This class represents a block in a world (grid) problem. Each block has an identifier (string for the
name),
and x and y coordinates representing its position in the grid (a Block[][] , see State and Node
classes).
The methods contained in here are standard getters and setters, plus a method to return the position
as
a string containing the coordinates
 */

public class Block {
```

```java
    private String name;            //tile identifier: either A,B,C,X (agent) or Empty (any other tile)
    private int yPos;                 //x and y coordinates for position in the grid (at given state)
    private int xPos;

    public Block(String name, int xPos, int yPos){      //create new block with passed parameters
        this.name = name;
        this.xPos = xPos;
        this.yPos = yPos;
    }

    public String getName(){
        return this.name;
    }

    public int getYPos(){
        return this.yPos;
    }

    public int getXPos(){
        return this.xPos;
    }

    public void setXPos(int newXPos){          //methods to set new positions
        this.xPos = newXPos;
    }

    public void setYPos(int newYPos){
        this.yPos = newYPos;
    }

    public String getPosition(){        //retrieves position returned as coordinate string (eg (1,2) )
        return "(" + this.xPos + "," + this.getYPos() + ")";
    }

    //equals() and hashCode() created automatically
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        Block block = (Block) o;

        if (xPos != block.xPos) return false;
        if (yPos != block.yPos) return false;
        if (!name.equals(block.name)) return false;

        return true;
    }

    @Override
    public int hashCode() {
        int result = name.hashCode();
        result = 31 * result + yPos;
        result = 31 * result + xPos;
        return result;
    }
}
```

Move class (enum)

```java
/*
This enum represents all the possible moves the agent can take, which will be analysed/authorised or
rejected
in the Node/State classes
*/
```

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

Node class

```
import java.util.ArrayList;

/*
    This is another one of the basic 'infrastructure' classes of the project, representing a Node in
the search
 tree for the blocksworld puzzle. Each Node has a State (board configuration), heuristic estimated
cost to the
 end node for the heuristic search, a parent node and other characteristics
 */

public class Node implements Comparable {

    private Node parent;       //basic node characteristics
    private State state;
    private int pathCost;
    private int depth;
    private String direction;
    private int heuristic;      //holds the estimated heuristic cost of a node (to the finish node)

    public Node(Node parent, State state, String direction){      //non-root constructor
        this.parent = parent;
        this.state = state;
        if(parent != null){
            this.pathCost = parent.pathCost + 2;
            this.depth = parent.depth + 1;
        }
        this.direction = direction;
    }

    public Node(State state){            //root constructor
        this.state = state;
        this.depth = 0;
        this.pathCost = 0;
    }

    public Node getParent(){
        return parent;
    }

    public State getState(){
        return state;
    }

    public int getDepth(){
        return depth;
    }

    public int getCost(){
        return pathCost;
    }

    /*
    Method to create and return an ArrayList of Nodes representing the possible moves from this
instance of node.
```

```
    The x/y coordinates are checked against the array limits (for the world boundaries), then if the
move is
    feasible a new temporary node is created, in which the agent moves in that specific direction
(swapping positions
    with the tile that was in its place earlier). The new (neighbour) node is added to the arraylist
of possible
    moves, which will be retrieved by the search methods when calling this method.
     */

    public ArrayList<Node> checkMoves(){

        ArrayList<Node> moves = new ArrayList<Node>();

        if(state.agent.getYPos() > 0){

            State state1 = new State(this.state.a.getXPos(),this.state.a.getYPos(),
this.state.b.getXPos(),this.state.b.getYPos(),this.state.c.getXPos(),this.state.c.getYPos(),
this.state.agent.getXPos(),this.state.agent.getYPos(), this.state.gridDimension);

            Node temp = new Node(this,state1, "U");
            temp.state.moveAgent(Move.UP);
            moves.add(temp);
        }

        if(state.agent.getYPos() < state.gridDimension - 1){

            State state1 = new State(this.state.a.getXPos(),this.state.a.getYPos(),
this.state.b.getXPos(),this.state.b.getYPos(),this.state.c.getXPos(),this.state.c.getYPos(),
this.state.agent.getXPos(),this.state.agent.getYPos(),this.state.gridDimension);

            Node temp = new Node(this,state1, "D");
            temp.state.moveAgent(Move.DOWN);
            moves.add(temp);
        }

        if(state.agent.getXPos() > 0){

            State state1 = new State(this.state.a.getXPos(),this.state.a.getYPos(),
this.state.b.getXPos(),this.state.b.getYPos(),this.state.c.getXPos(),this.state.c.getYPos(),
this.state.agent.getXPos(),this.state.agent.getYPos(),this.state.gridDimension);

            Node temp = new Node(this,state1, "L");
            temp.state.moveAgent(Move.LEFT);
            moves.add(temp);
        }

        if(state.agent.getXPos() < state.gridDimension - 1){

            State state1 = new State(this.state.a.getXPos(),this.state.a.getYPos(),
this.state.b.getXPos(),this.state.b.getYPos(),this.state.c.getXPos(),this.state.c.getYPos(),
this.state.agent.getXPos(),this.state.agent.getYPos(),this.state.gridDimension);

            Node temp = new Node(this,state1, "R");
            temp.state.moveAgent(Move.RIGHT);
            moves.add(temp);
        }

        return moves;
    }

    @Override
    public boolean equals(Object o) {       //generated automatically
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        Node node = (Node) o;
        if (state != null ? !state.equals(node.state) : node.state != null) return false;

        return true;
    }

    @Override
    public int hashCode() {                  //generated automatically
        int result = (state != null ? state.hashCode() : 0);
        return result;
    }


    /*
    Method to display the path taken from the start node to the solution (in terms of directions
taken): an ArrayList
    of Strings for the directions is created, and while the parent of the node considered (this
instance) is not null,
    the direction taken to get to it from its parent is added to the list, and so on iteratively.
    A String[] is created by converting the AL and returned by the method.
     */

    public String[] displaySolution(){
        Node node = this;
        ArrayList<String> moves = new ArrayList<String>();

        while(node.parent != null){
            moves.add(0,node.direction);  //while parent isn't null, add direction from parent to this
            node = node.parent;
        }

        String[] result = (String[]) moves.toArray(new String[moves.size()]);
        return result;
    }


    /*
    Method to estimate heuristic cost from current node to end node, by summing path distance for each
tile from current
    (this) node to end node and further summing the depth of the current node. The agent cost has been
commented out to
    make the heuristics accurate and consistent, otherwise data obtained was deemed inconsistent.
     */

    public void getCostEstimate(Node target){            //returns heuristic estimate

        int aCostX = Math.abs(target.getState().a.getXPos() - getState().a.getXPos());
        int aCostY = Math.abs(target.getState().a.getYPos() - getState().a.getYPos());
        int bCostX = Math.abs(target.getState().b.getXPos() - getState().b.getXPos());
        int bCostY = Math.abs(target.getState().b.getYPos() - getState().b.getYPos());
        int cCostX = Math.abs(target.getState().c.getXPos() - getState().c.getXPos());
        int cCostY = Math.abs(target.getState().c.getYPos() - getState().c.getYPos());
        //int agentCostX = Math.abs(target.getState().agent.getXPos() - getState().agent.getXPos());
        //int agentCostY = Math.abs(target.getState().agent.getYPos() - getState().agent.getYPos());

        this.heuristic = aCostX + aCostY + bCostX + bCostY + cCostX + cCostY + getDepth();
    }


    /*
    compareTo method overriding the one in the Object class, returns an int depending on the
comparison result
     */

    public int compareTo(Object node){
        if(heuristic < ((Node) node).heuristic) return -1;
        else if(heuristic == ((Node) node).heuristic) return 0;
        else return 1;
    }
}
```

State class

```
/*
This class represents a state in the blocksworld (ie, a specific configuration of tiles in the board).
It has methods to make the agent perform a move and other variables used by other classes.
 */

public class State {

    private Block[][] blocks;   //grid, array of arrays of Blocks [ie rows and columns for a 2D space]
    public int gridDimension;       //block array (grid) dimension (ie, if 4, then it will be 4x4)
    private int maxArrDim;              //max array actual dimension (will be set to grid dimension -1)
    public Block agent, a, b, c;             //blocks for tiles A,B,C and agent.


    public State(int xa, int ya, int xb, int yb, int xc, int yc, int xAg, int yAg, int worldSize)
throws IllegalArgumentException{

        this.gridDimension = worldSize;     //set the max grid dimension from the parameter passed
        this.maxArrDim = worldSize - 1;

        /*
        check for illegality of arguments: if x/y positions smaller than 0 or bigger than max array
(grid) positions,
        check if any tiles (including the agent) have been given same positions. Throw an Exception in
any case.
         */

        if(xa > maxArrDim || xb > maxArrDim || xc > maxArrDim || xAg > maxArrDim || ya > maxArrDim ||
yb > maxArrDim || yc > maxArrDim || yAg > maxArrDim ||
xa < 0 || xb < 0 || xc < 0 || xAg < 0 || ya < 0 || yb < 0 || yc < 0 || yAg < 0){

            throw new IllegalArgumentException("Error: one of the parameters for the tiles is invalid.
The minimum x/y position” + “for a tile is 0 and the maximum is " + maxArrDim);

        }

        else if((xa == xb && ya == yb) || (xb == xc && yb == yc) || (xa == xc && ya == yc)){

            throw new IllegalArgumentException("Error: different positions have to be specified for
the tiles. Please ensure that" + “the x/y parameters provided differ!");

         }

        else if((xAg == xa && yAg == ya) || (xAg == xb && yAg == yb) || (xAg == xc && yAg == yc)){

            throw new IllegalArgumentException("Error: the agent's position has to be different from
the tiles' ones. Please ensure" + "that the x/y parameters provided for the agent differ!");

        }

        else{     //if all correct, initialise and insert empty tiles and A/B/C/agent tiles in array

            blocks = new Block[gridDimension][gridDimension];
            agent = new Block("X",xAg,yAg);   //create the blocks, agent identified by the letter "X"
            a = new Block("A",xa,ya);
            b = new Block("B",xb,yb);
            c = new Block("C",xc,yc);

            for(int i=0;i<gridDimension;i++){
                for(int j=0;j<gridDimension;j++){
                    setPosition(i,j,new Block("Empty",i,j));        //set all tiles to be empty tiles
                }
            }

            setPosition(xa,ya,a);           //set the positions of the tiles
            setPosition(xb,yb,b);
            setPosition(xc,yc,c);
            setPosition(xAg,yAg,agent);
        }
    }


    /*
        Method to set a block of the array to be at the passed coordinates and with the passed block
     */

    public void setPosition(int x, int y, Block block){
        blocks[x][y] = block;
    }

    public String toString(){           // will return cells' positions as a String

        return "The blocks are in the following positions: A - (" + a.getXPos() + "," + a.getYPos() +
")" +
                ", B - (" + b.getXPos() + "," + b.getYPos() + ")" +
                ", C - (" + c.getXPos() + "," + c.getYPos() + ")" +
                ", Agent (X) - (" + agent.getXPos() + "," + agent.getYPos() + ")";

    }


    /*
        Method to compare the current state to the goal state (positions of A,B,C,Agent being fixed)
by comparing the
        Strings representing those tiles' positions
     */

    public boolean compareTo(State state){
        return (this.a.getPosition().equals(state.a.getPosition())
                && this.b.getPosition().equals(state.b.getPosition())
                && this.c.getPosition().equals(state.c.getPosition())
                && this.agent.getPosition().equals(state.agent.getPosition()));
    }

     /*
    Method to compare the current state to the goal state (where the agent can end anywhere, with only
A/B/C being fixed), works similarly to compareTo(State state).
     */

    public boolean equalToGoal(State goal){
        return (this.a.getPosition().equals(goal.a.getPosition())
                && this.b.getPosition().equals(goal.b.getPosition())
                && this.c.getPosition().equals(goal.c.getPosition()));
    }

    /*
    Method to make the agent move. It analyses the passed move, then utilises a switch block to
determine the action to
    take. For each move, if it is permitted (ie, the agent is within the boundaries where that move
can be achieved) then
    a temporary block is created with the position of the tile the agent has to shift with. Both the
agent and the other
    tile are shifted and their positions are set. True is returned if the move is possible, false
otherwise.
     */

    public boolean moveAgent(Move move) throws IllegalArgumentException {

        switch(move){
            case UP:
                if(agent.getYPos() == 0) return false;
                else{
                    int newYPos = agent.getYPos() - 1;
                    Block old = blocks[agent.getXPos()][newYPos];
                    blocks[agent.getXPos()][agent.getYPos()] = old;
                    old.setYPos(agent.getYPos());
                    blocks[agent.getXPos()][newYPos] = agent;
                    agent.setYPos(newYPos);

                    return true;
                }
            case DOWN:
                if(agent.getYPos() == maxArrDim) return false;
                else{
                    int newYPos = agent.getYPos() + 1;
                    Block old = blocks[agent.getXPos()][newYPos];
                    blocks[agent.getXPos()][agent.getYPos()] = old;
                    old.setYPos(agent.getYPos());
                    blocks[agent.getXPos()][newYPos] = agent;
                    agent.setYPos(newYPos);

                    return true;
                }
            case LEFT:
                if(agent.getXPos() == 0) return false;
                else{
                    int newXPos = agent.getXPos() - 1;
                    Block old = blocks[newXPos][agent.getYPos()];
                    blocks[agent.getXPos()][agent.getYPos()] = old;
                    old.setXPos(agent.getXPos());
                    blocks[newXPos][agent.getYPos()] = agent;
                    agent.setXPos(newXPos);

                    return true;
                }
            case RIGHT:
                if(agent.getXPos() == maxArrDim) return false;
                else{
                    int newXPos = agent.getXPos() + 1;
                    Block old = blocks[newXPos][agent.getYPos()];
                    blocks[agent.getXPos()][agent.getYPos()] = old;
                    old.setXPos(agent.getXPos());
                    blocks[newXPos][agent.getYPos()] = agent;
                    agent.setXPos(newXPos);

                    return true;
                }
        }
        return false;
    }


    /*
        Method used to print the board's state at any time (used for testing purposes only)

    public void performTests(){
        for(Block[] blockArr : blocks){
            for(Block block : blockArr){
                if(block != null){
                    System.out.println("Now in position " + block.getXPos() + "," + block.getYPos() +
" we have " + block.getName());
                }
                else{
                    System.out.println("NULL BLOCK");
                }
            }
        }
    }
     */

    @Override
    public boolean equals(Object o) {       //generated automatically
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        State state = (State) o;

        if (gridDimension != state.gridDimension) return false;
        if (maxArrDim != state.maxArrDim) return false;
        if (a != null ? !a.equals(state.a) : state.a != null) return false;
        if (agent != null ? !agent.equals(state.agent) : state.agent != null) return false;
        if (b != null ? !b.equals(state.b) : state.b != null) return false;
        if (c != null ? !c.equals(state.c) : state.c != null) return false;

        return true;
    }

    @Override
    public int hashCode() {                 //generated automatically
        int result = gridDimension;
        result = 31 * result + maxArrDim;
        result = 31 * result + (agent != null ? agent.hashCode() : 0);
        result = 31 * result + (a != null ? a.hashCode() : 0);
        result = 31 * result + (b != null ? b.hashCode() : 0);
        result = 31 * result + (c != null ? c.hashCode() : 0);
        return result;
    }
}
```

APPENDIX B
- Solution to a 4x4 grid [example output]-

Below is the output of the code running on a problem of difficulty 4 (grid of size 4x4):

```
BOARD OF SIZE 4x4NOW RUNNING...
A* COMPLETE = Solution path: [L, L, L, U, R, D, R, R, U, U, L, L, D, D, R, R]
BFS COMPLETE = Solution path: [U, L, D, L, L, U, R, D, R, U, U, L, D, D, R, R]
DFS COMPLETE = Solution path: [L, L, L, U, R, R, R, D, L, L, L, U, U, R, R, R, D, L, L, L, U, U, R, R, R, D, L, L, L, D, D, R, R, R, U, L, L,
L, D, R, R, R, U, U, L, L, L, D, D, R, R, R, U, U, L, L, L, D, R, R, R, U, U, L, L, L, D, R, R, R, D, L, L, L, D, R, R, R, U, U, L, L, L, D, R, R, R, U, L, L, L, D, R, R,
R, D, L, L, L, U, U, R, R, R, D, D, L, L, L, U, U, R, R, R, D, L, L, L, D, R, R, R, U, L, L, L, U, U, R, R, R, D, D, L, L, L, U, R, R, R, D, L, L, L, D, R, R, R, U, L, L,
L, U, U, R, R, R, D, L, L, L, D, D, R, R, R, U, L, L, L, D, R, R, R, U, L, L, L, D, R, R, R, U, L, L, L, D, R, R, R, U, U, L, L, L, D, R, R, R, U, U, L, L, L, D, D, R, R,
R, U, L, L, L, D, R, R, R, D, L, L, L, U, U, R, R, R, D, L, L, L, U, R, R, R, D, L, L, L, D, R, R, R, U, U, L, D, R, D, L, L, L, U, R, R, R, D, L, L, L, U, R, R, R, D, L,
L, L, U, R, R, R, D, L, L, L, U, R, R, R, D, L, U, R, D, L, L, L, U, R, R, R, D, L, L, L, U, R, R, R, D, L, L, L, U, R, R, R, D, L, U, R, D, L, L, L, U, R, R, R, D, L, L,
L, U, R, R, R, D, L, L, L, U, U, R, R, R, D, L, L, L, U, U, R, R, R, D, L, L, L, D, D, R, R, R, U, L, L, L, D, R, R, R, U, L, L, L, D, R, R, R, U, U, L, L, L, D, R, R, R,
U, U, L, L, L, D, R, R, R, D, L, L, L, D, R, R, R, U, U, L, L, L, D, D, R, R, R, U, U, L, L, L, D, R, R, R, D, L, L, L, U, U, R, R, R, D, D, L, L, L, U, R, R, R, U, U, L,
L, L, D, R, R, R, D, D, L, L, L, U, R, R, R, U, U, L, L, L, D, D, R, R, R, U, U, L, L, L, D, R, R, R, D, D, L, L, L, U, U, R, R, R, D, D, L, L, L, U, U, R, R, R, D, D, L,
L, L, U, U, R, R, R, D, L, L, L, U, U, R, R, R, D, L, L, L, D, R, R, R, U, U, L, L, L, D, D, R, R, R, U, L, L, L, D, D, R, R, R, U, L, L, L, U, U, R, R, R, D, L, L, L, D,
D, R, R, R, U, L, L, L, U, U, R, R, R, D, L, L, L, D, D, R, R, R, U, U, L, L, L, D, D, R, R, R, U, U, L, L, L, D, D, R, R, R, U, U, L, L, L, D, R, R, R, U, U, L, L, L, D,
R, R, R, D, L, L, L, U, U, R, R, R, D, D, L, L, L, U, R, R, R, D, D, L, L, L, U, U, R, R, R, D, D, L, L, L, U, R, R, R, U, U, L, L, L, D, R, R, R, D, D, L, L, L, U, U, R,
R, R, D, D, L, L, L, U, U, R, R, R, D, D, L, L, L, U, U, R, R, R, D, L, L, L, U, U, R, R, R, D, L, L, L, D, R, R, R, U, U, L, L, L, D, D, R, R, R, U, L, L, L, D, D, R, R,
U, R, D, L, L, L, U, U, R, R, R, D, D, L, L, L, U, U, R, R, R, D, L, L, L, U, R, R, R, D, L, L, L, D, R, R, R, U, U, L, L, L, D, D, R, R, R, U, L, L, L, U, U, R, R, R, D,
D, L, L, L, U, U, R, R, R, D, D, L, L, L, U, U, R, R, R, D, L, L, L, D, R, R, R, D, L, L, L, U, U, R, R, R, D, L, L, L, U, R, R, R, D, L, L, L, D, R, R, R, U, U, L, L, L,
D, D, R, R, R, U, U, L, L, L, D, R, R, R, D, L, L, L, U, R, R, R, U, U, L, L, L, D, D, R, R, R, U, L, L, L, D, R, R, R, D, L, L, L, U, R, R, R, U, U, L, L, L, D, R, R, R,
D, D, L, L, L, U, R, R, R, D, L, L, L, U, R, R, R, D, L, L, L, U, R, R, R, D, L, L, L, U, U, R, R, R, D, L, L, L, U, U, R, R, R, D, D, L, L, L, U, R, R, R, D, L, L, L, D,
R, R, R, U, U, L, L, L, D, R, R, R, U, L, L, L, D, R, R, R, D, L, L, L, U, U, R, D, R, R, D, L, L, L, U, R, R, R, D, L, L, L, U, R, R, R, D, L, L, L, U, R, R, R, D, L, L,
L, U, R, D, R, R, U, L, L, L, D, R, R, R, U, U, L, L, L, D, R, R, R, U, U, L, L, L, D, D, R, R, R, U, U, L, L, L, D, R, R, R, D, D, L, L, L, U, R, R, R, D, L, L, L, U, U,
R, R, R, D, D, L, L, L, U, U, R, R, R, D, L, L, L, U, U, R, R, R, D, L, L, L, D, R, R, R, D, L, L, L, U, U, R, R, R, D, L, L, L, U, R, R, R, D, L, L, L, D, R, R, R, U, U,
L, L, L, D, D, R, R, R, U, U, L, L, L, D, R, R, R, D, L, L, L, U, R, R, R, U, U, L, L, L, D, D, R, R, R, U, L, L, L, D, R, R, R, D, L, L, L, U, R, R, R, U, U, L, L, L, D,
R, R, R, D, D, L, L, L, U, R, D, R, R, U, U, L, L, L, D, R, R, R, U, U, L, L, L, D, R, R, R, D, D, L, L, L, U, U, R, R, R, D, D, L, L, L, U, U, R, R, R, D, D, L, L, L, U,
U, R, R, R, D, L, L, L, U, U, R, R, R, D, L, L, L, D, R, R, R, U, U, L, L, L, D, R, D, R, R, U, L, L, L, D, R, R, R, D, L, L, L, U, U, R, D, R, R, U, U, L, L, L, D, R, R,
R, D, D, L, L, U, R, R, U, U, L, L, L, D, R, R, R, D, D, L, L, L, U, U, R, R, R, D, D, L, L, L, U, U, R, R, R, D, D, L, L, L, U, U, R, R, R, D, L, L, L, U, U, R, R, R,
D, L, L, L, U, R, R, R, D, D, L, L, L, U, R, R, R, D, D, L, L, L, U, R, R, U, U, R, D, L, L, L, D, D, R, R, R, U, U, L, L, L, D, D, R, R, R, U, L, L, L, U, R, R, R, D, D,
L, L, L, U, R, R, R, D, L, L, L, U, R, R, R, U, U, L, L, L, D, D, R, R, R, U, U, L, L, L, D, D, R, R, R, U, U, L, L, L, D, R, R, R, D, L, L, L, U, U, R, R, R, D, D, L, L,
L, D, R, R, R, U, U, U, L, L, D, R, R, D, D, L, L, L, U, R, R, R, U, U, L, L, L, D, R, R, R, D, D, L, L, L, U, U, R, R, R, D, D, L, L, L, U, R, R, R, D, L, L, L, U, R, R,
R, U, L, L, L, D, D, R, R, R, U, U, L, L, L, D, D, R, R, R, U, L, L, L, U, R, R, R, D, L, L, L, U, U, R, R, R, D, L, L, L, D, R, R, R, U, U, L, L, L, D, R, R, R, D, L, L,
L, U, R, R, R, D, L, L, L, D, R, R, R, U, L, L, L, U, R, R, R, D, L, L, L, U, R, R, R, D, L, L, L, D, R, R, R, U, L, L, L, U, U, R, R, R, D, L, L, L, D, D, R, R, R, U, U,
L, L, D, R, R, U, U, L, L, L, D, D, R, R, R, U, L, L, L, D, R, R, R, D, L, L, L, U, U, R, R, R, D, L, L, L, D, R, R, R, U, L, L, L, D, R, R, R, U, L, L, L, U, U, R, R, R,
D, D, L, L, L, U, U, R, R, R, D, L, L, D, D, R, R, U, U, L, L, L, D, D, R, R, R, U, U, L, L, L, D, R, R, R, U, U, L, L, L, D, R, R, R, D, L, L, L, D, R, R, R, U, U, L, L,
L, D, R, R, R, U, L, L, L, D, R, R, R, D, L, L, L, U, U, R, R, R, D, D, L, L, L, U, U, R, R, R, D, L, L, L, D, R, R, R, U, L, L, L, U, U, R, R, R, D, D, L, L, L, U, R, R,
R, D, L, L, L, D, R, R, R, U, L, L, L, U, U, R, R, R, D, L, L, L, D, D, R, R, R, U, L, L, L, D, R, R, R, U, L, L, L, D, R, R, R, U, L, L, L, D, R, U, L, D, R, R, R, U, U,
L, L, L, D, D, R, R, R, U, U, L, L, L, D, R, R, R, U, U, L, L, L, D, R, R, R, D, L, L, L, D, R, R, R, U, U, L, L, L, D, R, R, R, U, L, L, L, D, R, R, R, D, L, L, L, U, U,
R, R, R, D, D, L, L, L, U, U, R, R, R, D, L, L, L, D, R, R, R, U, L, L, L, U, U, R, R, R, D, D, L, L, L, U, R, R, R, D, L, L, L, D, R, R, R, U, L, L, L, U, U, R, R, R, D,
L, L, D, R, R, D, L, L, L, U, R, R, R, D, L, L, L, U, R, R, R, D, L, L, L, U, R, R, R, U, U, L, L, L, D, D, R, R, R, U, L, L, L, D, R, R, R, D, L, L, L, U, U, R, R, D, R,
D, L, L, L, U, R, R, R, D, L, L, L, U, R, R, R, D, L, L, L, U, U, R, R, R, D, L, L, L, U, U, R, R, R, D, D, L, L, L, U, R, R, R, D, D, L, L, L, U, U, R, R, R, D, D, L, L,
L, U, R, R, R, D, L, L, L, U, R, R, R, U, U, L, L, L, D, D, R, R, R, U, U, L, L, L, D, R, R, R, D, D, L, L, L, U, R, R, R, D, L, L, L, U, R, R, R, U, U, L, L, L, D, R, R,
D, D, R, U, U, L, L, L, D, D, R, R, R, U, U, L, L, L, D, D, R, R, R, U, U, L, L, L, D, R, R, R, U, U, L, L, L, D, R, R, R, D, D, L, L, L, U, R, R, R, U, U, L, L, L, D, R,
R, R, D, D, L, L, L, U, U, R, R, R, D, D, L, L, L, U, U, R, R, R, D, D, L, L, L, U, U, R, R, R, D, L, L, L, U, U, R, R, R, D, L, L, L, U, R, R, R, D, L, L, L, D, D, R, R,
R, U, U, L, U, R, D, L, L, L, D, D, R, R, R, U, U, L, L, L, D, D, R, R, R, U, L, L, L, U, U, R, R, R, D, L, L, L, D, R, R, R, D, L, L, L, U, U, R, R, R, D, L, L, L, U, R,
R, R, D, L, L, L, D, R, R, R, U, U, L, L, L, D, D, R, R, R, U, U, L, L, L, D, R, R, R, U, U, L, L, L, D, D, R, R, R, U, L, L, L, D, D, R, R, R, U, L, L, L, U, U, R, R, R,
D, D, L, L, L, U, U, R, R, R, D, D, L, L, L, U, R, R, R, D, L, L, L, U, R, R, R, D, D, L, L, L, U, R, R, R, D, L, L, L, U, R, R, R, U, U, L, L, L, D, R, R, R, D, D, L, L,
L, U, R, R, R, D, L, L, L, U, R, R, R, U, U, L, L, L, D, R, R, R, D, D, L, L, L, U, U, R, R, R, D, L, L, L, D, R, R, R, U, U, L, L, L, U, R, R, R, D, D, L, L, L, D, R, R,
R, U, U, L, L, L, D, R, R, R, D, L, L, L, U, U, R, R, D, R, U, L, L, L, D, D, R, R, R, U, U, U, L, L, L, D, R, R, R, D, L, L, L, U, R, R, R, D, L, L, L, D, R, R, R, U, U,
U, L, L, L, D, R, R, R, D, D, L, L, L, U, R, R, R, U, U, L, L, L, D, R, R, R, D, D, L, L, L, U, R, R, R, U, U, L, L, L, D, R, R, R, D, D, L, L, L, U, R, R, R, U, U, L, L,
L, D, R, R, R, D, L, L, L, D, R, R, R, U, U, L, L, L, D, D, R, R, R, U, U, L, L, L, D, R, R, R, D, L, L, L, U, U, R, R, R, D, D, L, L, L, U, R, R, R, U, U, L, L, L, D, D,
R, R, R, U, U, L, L, L, D, D, R, R, R, U, U, L, L, L, D, R, R, R, D, D, L, L, L, U, U, R, R, R, D, D, L, L, L, U, U, R, R, R, D, D, L, L, L, U, U, R, R, D, D, R, U, L,
L, L, D, R, R, R, U, U, L, L, L, D, R, R, R, U, U, L, L, L, D, R, R, R, D, D, L, L, L, U, R, R, R, D, L, L, L, U, R, R, R, D, L, L, L, U, U, R, R, R, D, L, L, L, U, U, R,
R, D, R, D, L, L, L, U, U, R, R, R, D, L, L, L, D, R, R, R, D, L, L, L, U, U, R, R, R, D, L, L, L, U, R, R, R, D, D, L, L, L, U, R, R, R, D, L, L, L, U, R, R, R, U, U, L,
L, L, D, D, R, R, R, U, U, L, L, L, D, R, R, R, D, D, L, L, L, U, R, R, R, D, L, L, L, U, R, R, R, D, L, L, L, U, U, R, R, R, D, L, L, L, D, R, R, R, U, U, L, L, L, D, R,
R, R, D, L, L, L, U, U, R, R, R, D, L, L, L, U, U, R, R, R, D, D, L, L, D, R, U, L, L, D, R, R, R, U, U, L, L, L, D, R, R, R, U, L, L, L, D, D, R, R, R, U, U, L, L, L, D,
R, R, R, U, L, L, L, D, R, R, R, U, U, L, L, L, D, R, R, R, D, L, L, L, D, R, R, R, U, U, L, L, D, R, R, D, L, L, L, U, U, R, R, R, D, L, L, L, U, U, R, R, R, D, D, L, L,
L, D, R, R, R, U, L, L, L, U, U, R, R, R, D, L, L, L, D, D, R, R, R, U, L, L, L, U, U, R, R, R, D, L, L, L, D, D, R, R, R, U, L, L, L, U, U, R, R, R, D, D, L, L, L, U, U,
R, R, R, D, D, L, L, L, U, U, R, R, R, D, D, L, L, L, U, R, R, R, D, D, L, L, L, U, R, R, R, U, L, L, L, D, D, R, R, R, U, U, L, L, L, D, R, R, R, D, L, L, L, U, U, R, R,
R, D, D, L, U, R, D, L, L, L, U, R, R, R, U, U, L, L, L, D, D, R, R, R, U, U, L, L, L, D, R, R, R, D, D, L, L, L, U, R, R, R, U, U, L, L, L, D, R, R, U, L, L, D, D, R, R,
D, L, L, U, R, R, R, D, L, L, L, U, U, R, R, R, D, D, L, L, L, U, U, R, R, R, D, L, L, L, U, U, R, R, R, D, L, L, L, D, D, R, R, R, U, L, L, D, R, R, U, U, L, L, L, D, D,
R, R, R, U, U, L, L, L, D, D, R, R, R, U, U, L, L, L, D, R, R, R, U, U, L, L, L, D, R, R, R, D, L, L, L, U, U, R, R, R, D, D, L, L, L, U, R, R, R, D, D, L, L, L, U, R, R,
R, U, U, L, L, L, D, R, R, R, D, D, L, L, L, U, R, R, R, U, U, L, L, L, D, R, R, R, D, D, L, L, L, U, U, R, R, R, D, D, L, L, L, U, U, R, R, R, D, D, L, L, L, U, U, R, R,
R, D, L, L, L, U, U, R, R, R, D, L, L, L, D, R, R, R, U, U, L, L, L, D, D, D, R, R, R, U, U, L, L, L, D, D, R, R, R, U, L, L, L, U, U, R, R, R, D, D, L, L, L, U, U, R, R,
R, D, D, L, L, L, U, U, R, D, R, R, D, D, L, U, R, D, L, L, L, U, U, R, R, R, D, D, L, L, L, U, U, R, U, L, D, R, R, R, D, L, L, L, D, R, R, R, U, L, L, L, U, U, R, R,
R, D, D, L, L, L, U, U, R, R, R, D, D, D, L, L, L, U, U, R, R, R, D, L, L, L, D, R, R, U, L, L, U, U, R, R, R, D, L, L, L, D, D, R, R, R, U, L, L, L, U, U, R, R, R, D, L,
L, L, D, D, R, R, R, U, L, D, R, U, L, L, D, R, R, U, U, L, L, L, D, D, R, R, R, U, U, L, L, L, D, D, R, R, R, U, U, L, L, L, D, R, R, R, U, U, L, L, L, D, R, R, R, D, L,
L, L, U, U, R, D, R, R, D, D, L, L, L, U, U, R, R, U, R, D, L, L, L, D, R, R, U, L, L, D, R, R, R, U, L, L, L, D, D, R, R, R, U, U, L, L, L, D, D, R, R, R, U, L, L, L, U,
U, R, R, R, D, L, L, L, D, D, R, R, R, U, L, L, L, U, U, R, R, R, D, L, L, L, D, D, R, R, R, U, U, L, L, L, D, D, R, R, R, U, U, L, L, L, D, D, R, R, R, U, U, L, L, L, D,
R, R, R, U, U, L, L, L, D, R, R, R, U, L, L, L, D, R, R, R, D, L, L, L, U, U, R, R, R, D, D, L, L, L, U, U, R, R, R, D, D, D, L, L, L, U, U, R, R, R, D, D, L, L, L, U, R,
R, R, U, U, L, L, L, D, R, R, R, D, D, L, L, L, U, R, R, R, U, U, L, L, L, D, D, R, R, R, U, U, L, L, L, D, R, R, R, D, D, L, L, L, U, U, R, R, R, D, D, L, L, L, U, U, R,
R, R, D, D, L, L, L, U, U, R, R, R, D, L, L, U, U, R, R, D, L, L, L, D, D, R, R, R, U, U, L, L, U, R, R, D, L, L, U, R, R, D, L, L, L, D, D, R, R, R, U, U, L, L, L, D, D,
R, R, R, U, U, L, L, L, D, D, R, R, R, U, L, L, L, U, U, R, R, R, D, D, L, L, L, U, U, R, R, R, D, D, L, L, L, U, R, U, L, D, R, R, D, D, R, U, L, L, L, D, R, R, R, U,
U, L, L, L, D, R, R, R, U, U, L, L, D, R, R, D, D, L, L, L, U, R, R, R, D, L, L, L, U, R, R, R, D, L, L, L, U, U, R, R, R, D, L, L, L, U, U, R, R, D, R, D, L, L, L, U, U,
R, R, R, D, L, L, L, D, D, R, R, R, U, U, U, L, L, L, D, R, R, R, D, L, L, L, D, R, R, R, U, U, L, L, L, U, R, R, R, D, D, D, L, L, L, U, R, D, L, U, U, R, D, R, R, U,
U, L, L, L, D, R, R, R, D, L, L, L, U, R, R, R, D, D, L, L, L, U, R, R, R, U, L, L, L, D, D, R, R, R, U, U, U, L, L, L, D, D, R, R, R, D, L, L, L, U, U, R, R, R, D, L, L,
L, D, R, R, R, U, L, L, L, U, U, R, R, R, D, D, L, L, L, U, U, R, R, R, D, D, L, L, U, U, L, D, R, D, R, R, D, L, L, L, U, U, R, R, R, D, L, L, L, U, U, R, D, L, D, D,
R, R, R, U, U, L, L, L, U, R, R, R, D, L, L, L, D, R, D, R, R, U, L, L, L, U, R, R, R, D, D, L, L, L, U, U, R, R, R, D, L, L, L, D, R, R, R, U, U, L, D, R, D, L, L, L, U,
R, R, R, U, U, L, L, L, D, D, R, R, R, U, U, L, L, L, D, D, D, R, R, R, U, U, L, L, L, D, R, R, R, U, U, L, L, L, D, R, R, R, D, D, L, L, L, U, U, R, R, R, D, L, L, L, D,
R, R, R, U, U, L, L, L, D, D, R, R, R, U, L, L, L, U, U, R, R, R, D, D, D, L, L, L, U, U, R, R, R, D, D, L, L, L, U, U, R, R, R, D, L, L, L, U, U, R, R, R, D, D, L, L,
U, R, R, D, L, L, L, U, R, R, R, D, D, L, L, L, U, R, R, R, U, U, L, L, L, D, R, D, L, D, R, R, R, U, U, L, L, L, D, R, R, R, U, U, L, L, L, D, R, R, R, D, D, L, L, L, U,
U, R, R, R, D, D, L, L, L, U, R, R, R, U, U, L, L, L, D, R, R, R, D, D, L, L, L, U, R, R, D, L, L, U, U, R, R, R, D, D, L, L, L, U, U, R, U, R, R, D, D, L, L, L, U, R,
R, R, D, D, L, L, L, U, U, R, R, R, D, L, L, L, D, R, R, R, U, U, L, L, L, D, D, R, R, R, U, L, L, L, U, U, R, R, R, D, D, L, L, L, U, U, R, R, R, D, D, L, L, L, U, U, R,
R, R, D, L, L, L, D, D, R, R, R, U, L, L, L, U, R, R, R, D, D, L, L, L, U, U, R, R, R, D, D, L, L, L, U, R, D, R, R, U, L, L, L, U, U, R, R, R, D, D, D, L, L, L, U, U, R,
R, R, D, L, L, L, U, U, R, R, D, R, D, L, L, L, U, U, R, R, R, D, D, L, L, L, U, U, R, R, R, D, L, D, L, L, U, U, R, R, R, D, L, L, L, U, R, R, R, D, L, L, L, D, R, R, R,
U, U, L, L, L, D, R, R, R, U, L, L, L, D, D, R, R, R, U, L, L, U, R, R, D, D, L, L, L, U, U, R, R, R, D, L, L, L, D, D, R, R, R, U, U, L, L, L, D, D, R, R, R, U, U, L, L,
L, D, D, R, R, R, U, L, L, L, U, U, R, R, R, D, L, L, L, D, R, R, R, D, L, L, L, U, U, R, R, R, D, L, L, L, U, R, R, R, D, L, L, L, D, R, R, R, U, U, L, L, L, D, D, R, R,

R, U, U, L, L, L, D, R, R, R, D, L, L, L, U, R, R, R, U, U, L, L, L, D, D, R, R, R, U, U, L, L, L, D, D, R, R, R, U, L, L, L, D, R, R, R, D, L, L, L, U, R, R, R, D]
IDS COMPLETE - Solution path: [L, L, L, U, R, D, R, R, U, L, U, L, D, D, R, R]
```