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

Roshan Patel (28302796)

28/11/2019

Assignment: Search methods
“Blocksworld tile puzzle”

Module Code: COMP2208

A report submitted for the award of
Bsc Computer Science (G400)

“I am aware of the requirements of good academic practice, and the potential penalties for any breaches”.

Approach

Representing the board

To represent the Blocksworld Tile Puzzle board, I created a BoardNode.java class that stores a local variable 'grid' which represents the state of the board as a 2-dimensional character array 'char[][]'.

Making the dimensions scalable

As an extension to make this problem scalable, the constructor for BoardNode takes an integer value 'rowsAndColumns', which is used as the dimension of the board. The start state of the board is created relative to the integer value given, so the board can be generated for any nxn dimension given rather than just the 4x4 representation shown in the problem. The location of the agent is also stored in its own local variable 'agent' of custom type 'intPair', explained below.

Agent representation – 'intPair'

As java does not have in-built implementation for tuple representation, I created a local class called 'intPair'. This class functions as a pair tuple storage method. It has a constructor which takes an int for the respective row and column locations, and stores it locally. It also has the relevant setter and getter methods for retrieving and updating the stored values.

Movement method

All valid possible movements for this puzzle include up, down, left, right. I created a 'movement' method which returns a Boolean to represent this. So for example if the agent is in the far right column of the 2d array representation, the Boolean method to move the agent 'right' would return false. So for each movement, if its condition to move in a particular direction is met, it makes the necessary array changes, returns true, and updates this in the local 'grid' variable, which (to reiterate) is type 'char[][]'.

Generate children method

This is the method for generating the children using the agent location for a particular board configuration. It has a return type of 'ArrayList<BoardNode>'. This method creates 4 new BoardNode 'child' objects, which are duplicates of the 'parent' of the children that are to be generated; one for each potential movement 'u' 'd' 'l' 'r' (up, down, left, right).

If the Boolean for the movement returns true, the 'child' is appended to the ArrayList of type <BoardNode> called 'children'. After all 4 movements have been tried, the resulting ArrayList is returned.

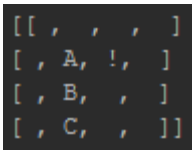
The depth of a child is calculated by taking the depth of its parent and incrementing it by 1. Note: The root node is instantiated at a depth of 0.

*I experienced one issue when trying to duplicate the parent BoardNode. Because it is an object, it only duplicated the reference pointer to the object. I overcame this by creating an overloaded constructor for my BoardNode class, which takes a BoardNode object as a parameter. It then copies each element from the 2d array stored in the BoardNode object using 'for' loops, and stores it in the local 'char[][] grid' variable. Once again, to make this scalable for any nxn board extension, the for loops were written with grid.length as a reference point.

I did the same thing for the 'agent location' using the get methods from my custom 'intPair' class.

Goal state

I set the goal state to exactly what was shown on the specification, which is A, B, and C such that all three are in the second column of the grid, C appears on the bottom row of the column, followed by B one tile above it and C one tile above B. Shown in the grid below. I assumed that the location of the agent '!' is irrelevant as detailed in the spec.



Breadth First Search (BFS)

My breadth first search class utilises a linked-list to represent the first in first out queue used in breadth first search. The constructor takes the root node and checks if it is a goal state. It then calls a recursive function which pops the node and creates the children in the order "Up, Down, Left, Right" and pushes all valid states to the queue (fringe). These states are checked in the order that they are put into the queue and and their respective children are also generated until a goal node is reached. The path to the node is printed using an "Arrays.deepToString" method to graphically accurately depict the board to the reader, along with the depth and the number of nodes generated to reach that node.

Depth First Search (DFS)

In my depth first search class I simulate a stack by using a linked list, where I utilise the poplast() method to pop the last element in the queue to effectively act as a stack (fringe). I encountered a problem where the moves made by the agent were constantly back and forth in an infinite loop, and to counter this, I randomized the order of the generated children. This search was also implemented recursively, and a path to the goal state is printed using the same methods as explained in my breadth first search above.

Iterative deepening depth first search (IDDFS)

Here I also simulate a stack by using a linked list, where I utilise the poplast() method to pop the last element in the queue to effectively act as a stack. Here I introduced a local variable called 'depthLimit', which is instantiated at 1. I also store the start state of the problem as the variable 'rootNode' to use as a restarting point for each depth iteration. When the depth limit is reached and my fringe (queue) is empty, I increment the depth limit by 1 and recursively restart the search with the locally stored root node.

A* Heuristic Search.

For my A* search heuristic, I calculated the Manhattan Distance (measuring the absolute x and y distances from the goal state), and added this to the node depth to generate an evaluation function 'cost'. Thus making the search A* rather than just greedy best first search. I tried using Java's in-built priority queue, however, I had difficulty accessing and comparing the cost elements associated with each step. I resolved this issue by creating my own BoardNodePriorityQueue class, which functions similar to a normal priority queue, with the additional feature that when it pops an element, it searches the ArrayList of BoardNodes to find the one with the lowest associated cost, and pops that one.

Evidence of Search methods

Start state for testing

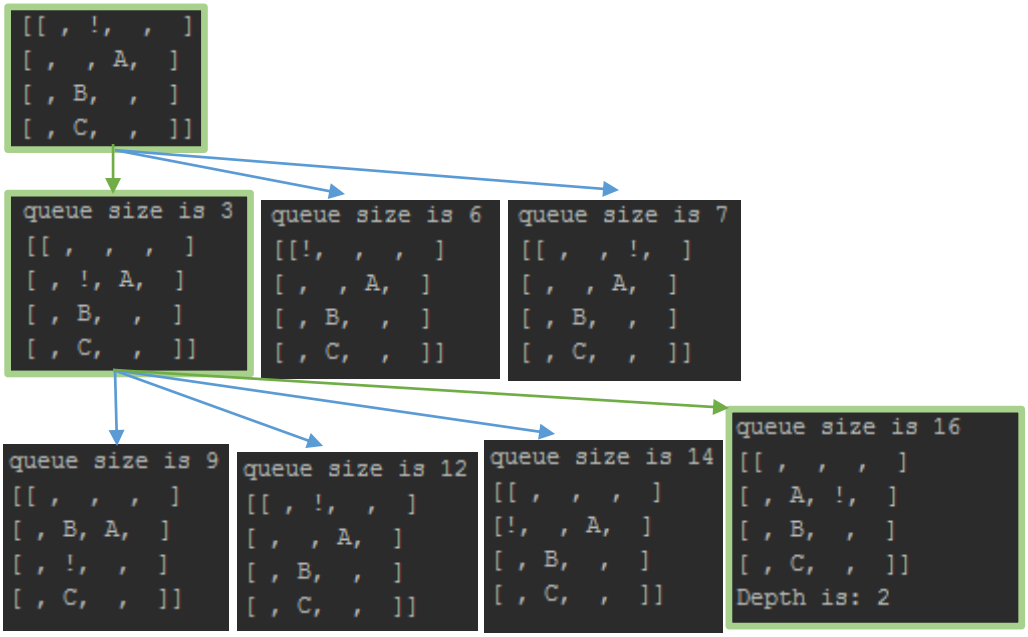
```
[[ , !,  ,  ]
[ ,  , A,  ]
[ , B,  ,  ]
[ , C,  ,  ]]
```

Goal State for testing

```
[[ ,  ,  ,  ]
[ , A, !,  ]
[ , B,  ,  ]
[ , C,  ,  ]]
```

The above start state and goal state is to be used for testing. I adjusted the start state to control the problem difficulty to depth level 2, and make the path to a solution visibly clear below. All the classes have been submitted, and the code for each class is listed in the appendices.

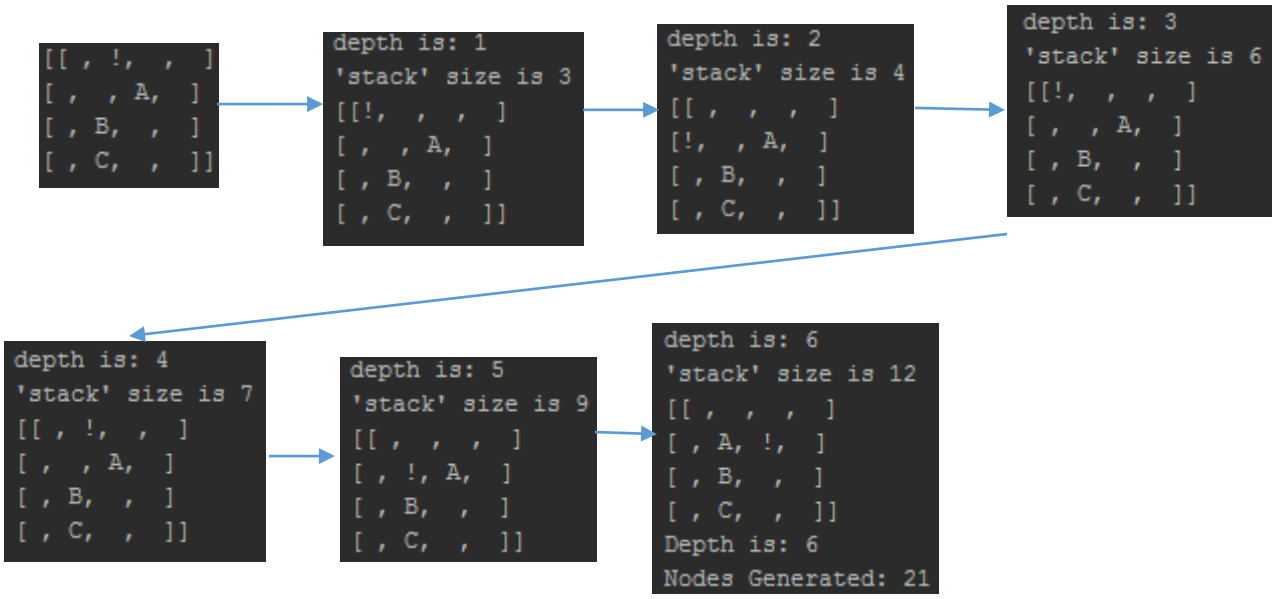
Breadth First Search (BFS)



The states with the green arrows and borders represent the optimal path to the goal state.

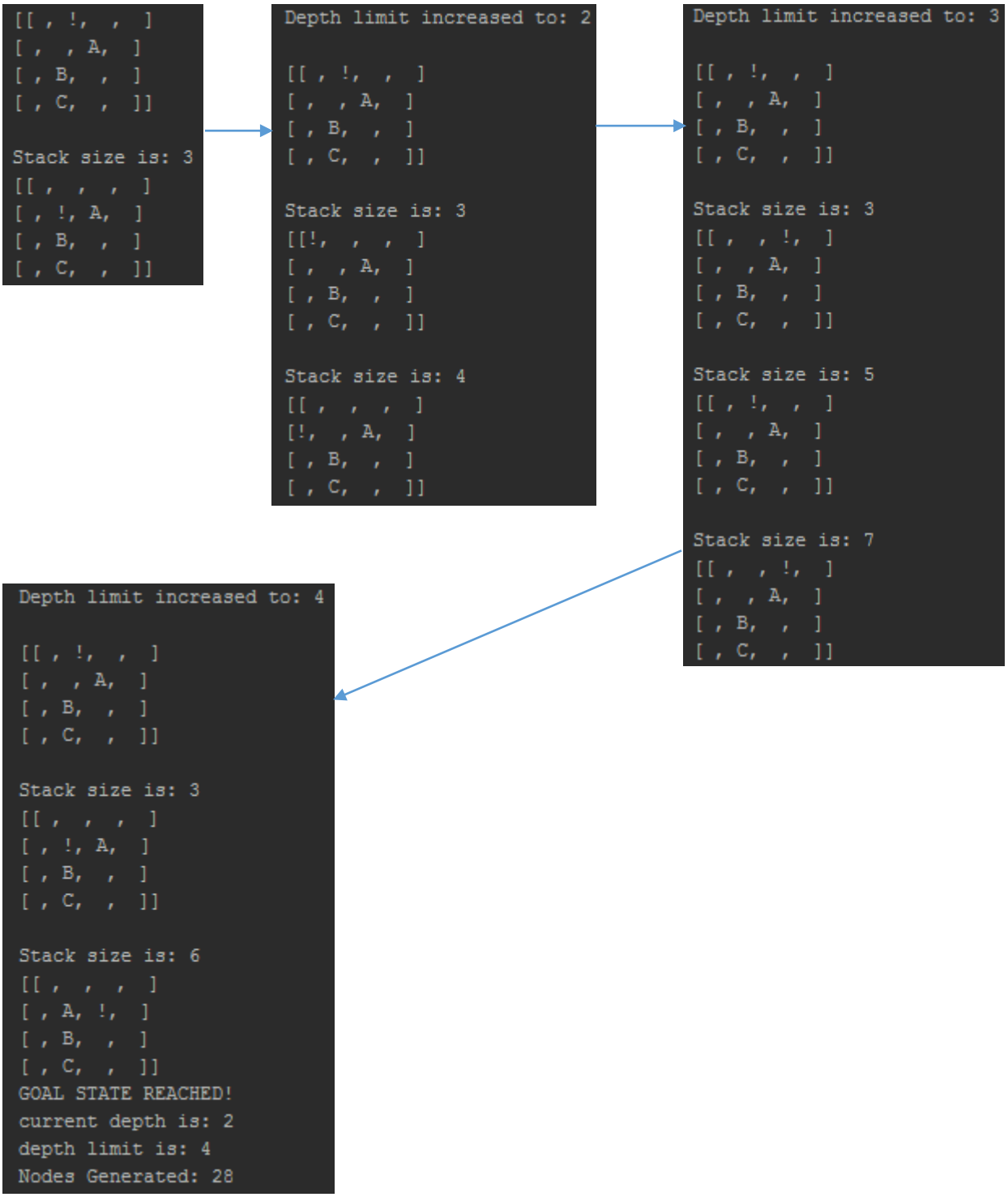
Depth First Search (DFS)

For depth first search, the search often went in a completely wrong direction, so the search was rerun until a solution that sufficiently demonstrated the correct implementation of the search was produced:



Iterative deepening depth first search (IDDFS)

Similar to depth first search, the IDDFS search was re-run until a solution that sufficiently demonstrated the correct implementation was reached. To reiterate, the start depth is 0 and the starting depth limit is 1.



```
[[ ,  !,  ,  ]
[ ,  ,  A,  ]
[ ,  B,  ,  ]
[ ,  C,  ,  ]]

Manhattan distance is: 1
Depth is: 1
Cost of next step is: 2
Priority Queue size is: 3
[[ ,  ,  ,  ]
[ ,  !,  A,  ]
[ ,  B,  ,  ]
[ ,  C,  ,  ]]

Manhattan distance is: 1
Depth is: 1
Cost of next step is: 2
Priority Queue size is: 6
[[ ,  ,  !,  ]
[ ,  ,  A,  ]
[ ,  B,  ,  ]
[ ,  C,  ,  ]]

Manhattan distance is: 1
Depth is: 1
Cost of next step is: 2
Priority Queue size is: 8
[[!,  ,  ,  ]
[ ,  ,  A,  ]
[ ,  B,  ,  ]
[ ,  C,  ,  ]]

Manhattan distance is: 0
Depth is: 2
Cost of next step is: 2
Priority Queue size is: 9
[[ ,  ,  ,  ]
[ ,  A,  !,  ]
[ ,  B,  ,  ]
[ ,  C,  ,  ]]
Depth is: 2
Nodes Generated: 16
```

For each step the node with the lowest cost is popped. The total cost is broken down into the Manhattan distance and the depth, which is summed together to produce the evaluation function of ‘cost’. The number of BoardNode objects in the priority queue is also printed.

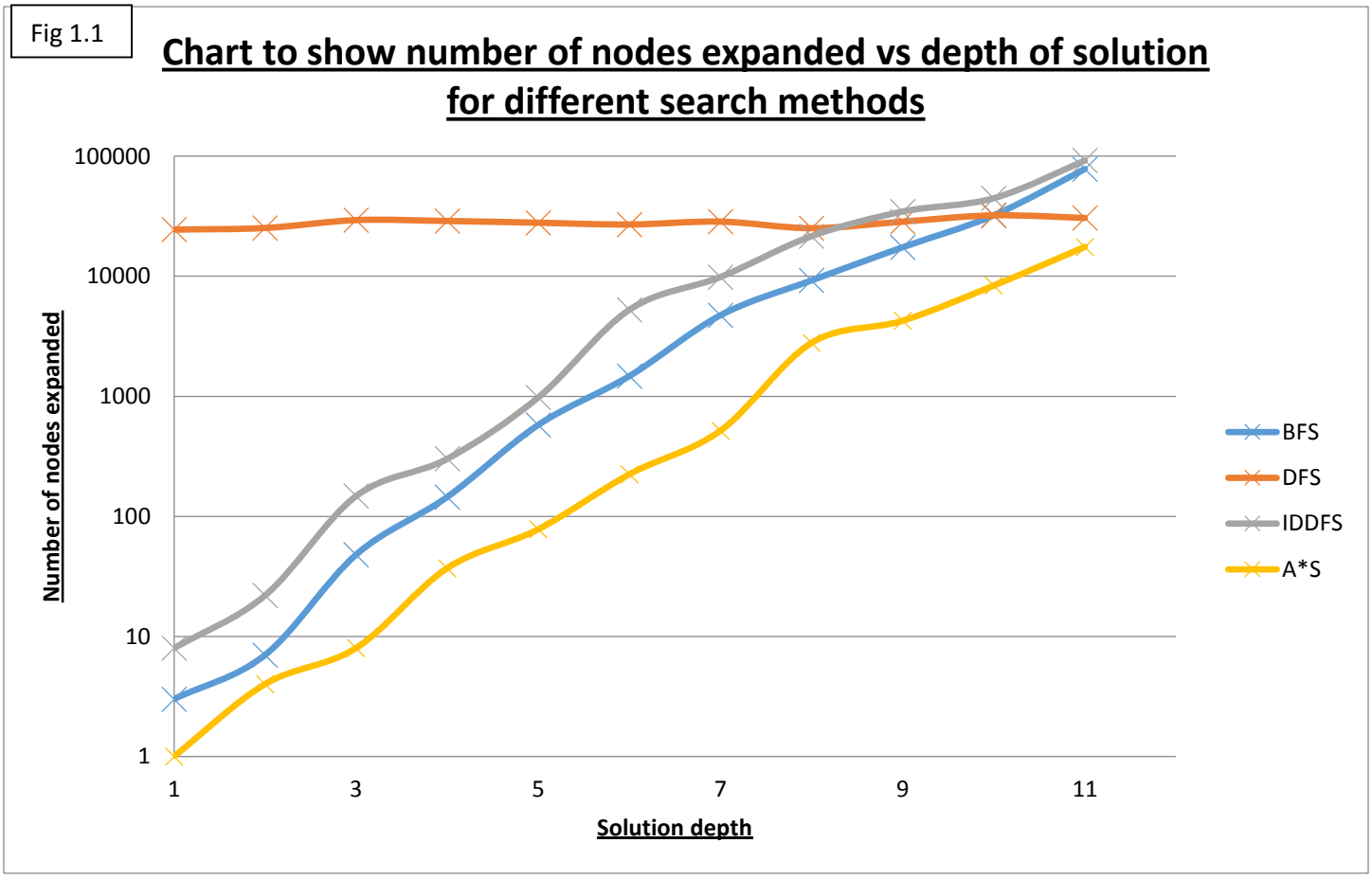
As this example is rather short and not fully demonstrative, the output for a solution that is at depth 4 using the Manhattan A* search is shown in the appendices.

Scalability study

To analyse the behaviour of each search method, I iteratively increased the depth of the solution, as suggested on the specification. For depth first search, iterative deepening search, and even BFS (due to the randomisation of the nodes), the search was run 150 times in order to establish an average number of nodes expanded value at each depth. This number was chosen because beyond it, the average value was observed to not change much. The A* search was also run 150 times for empirical equality, however, it was observed that the number of nodes expanded was always the same.

To make clear, as confirmed by the professor, the **number of nodes 'expanded' is defined as the number of nodes generated**. As the number of nodes expanded was mostly exponentially increasing, the axis for it is represented below at log base 10 to ease readability. If it interests the reader, a chart with linear axis is shown in the appendices. Each search was run up to a depth of 11, after which point most searches would produce a 'stack overflow error'. This is because for each depth the possible nodes generated increases by 4^d, where d is the depth.

Fig 1.1

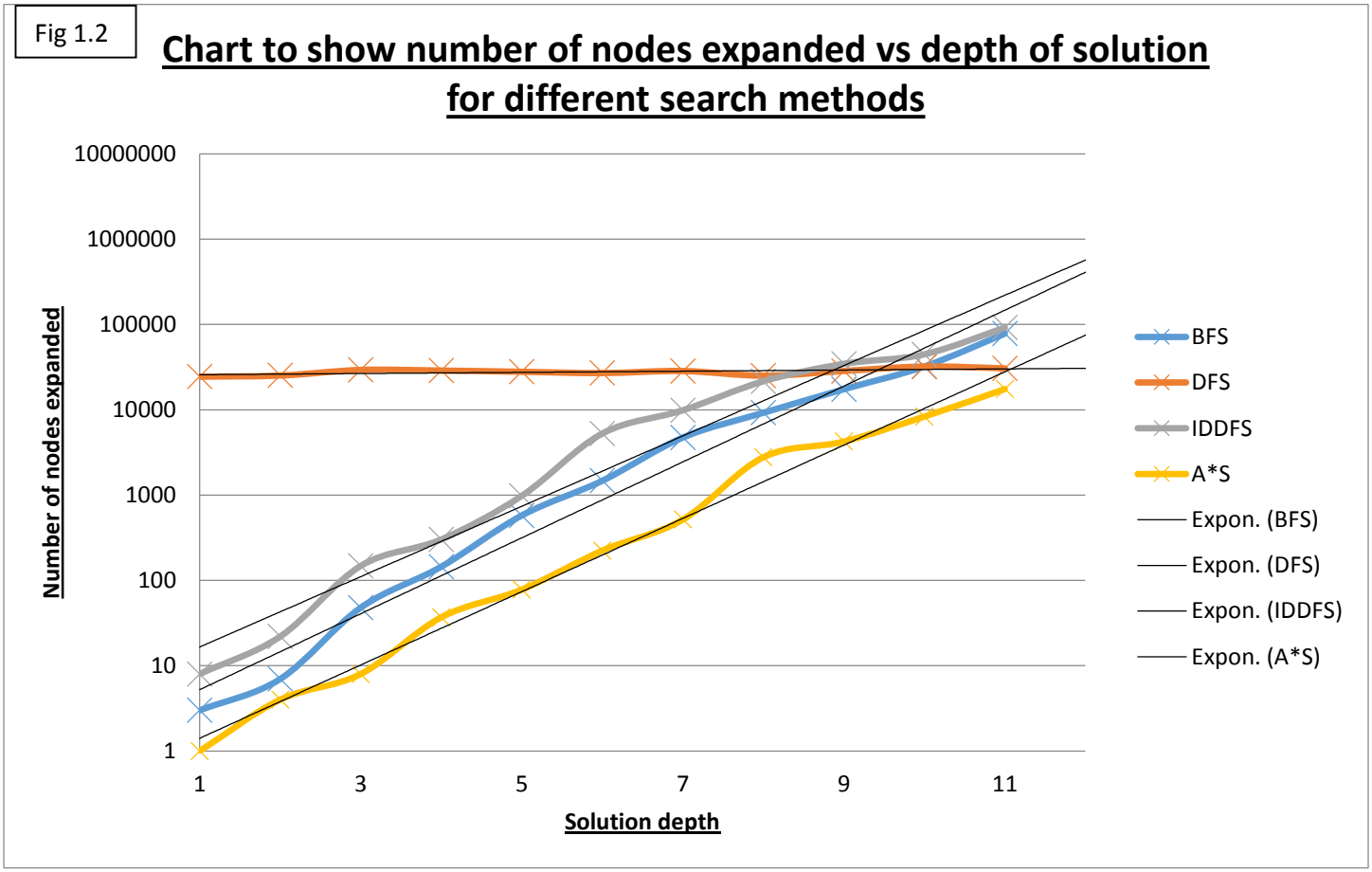


From fig 1.1, it is immediately evident that the most efficient search method of the 4 implemented was A* search, as it consistently has better number of nodes expanded. Breadth-first search was the next best, although considerably worse than A* search, due to the lack of an evaluation function. Iterative deepening is arguably the worst performing case, as it is consistently the worst after the depth 9, and although the difference may seem marginal on the graph above, once the log base 10 axis is taken into consideration, it can be seen that it exponentially gets a lot worse than breadth first. This is most definitely due to the nature of iterative deepening having to start from the root node again once the max depth is incremented. Depth first search was consistent throughout each solution depth level. The reason it had such a

high number of nodes expanded even at the lowest solution depths is probably due to the nature of it being able to go in the completely wrong direction and get further and further away from a solution.

The start state given in the spec found a solution at depth 15. The data suggests that the depth first search method, on average, would be the fastest way to get to a solution. Indeed, it should be the case that for most configurations of the blocksworld tile puzzle, depth first search would be the preferred method of search choice, as there exists far more configurations beyond the depth level of 11.62 (4.d.p), at which point the trend-lines generated in fig 1.2 below, suggest depth first search surpasses the other search methods.

Fig 1.2



The depth intersection 11.62, was calculated by intersected the trend-lines for Depth first search and A* search. A depth of 11.62 seemed quite low, so for some additional extensions I added 2 new heuristics and compared them to this existing one to see if the number at which depth first search surpasses other search methods could be increased, explained further after the evaluation.

Evaluation

From a data collection perspective, a huge limiting factor for the solution depth at which I could run the simulation was heap space, which often led to me experiencing a stack overflow error. I found a command "-Xms512m -Xmx1152m -XX:MaxPermSize=256m -XX:MaxNewSize=256m", which could increase the heap size, but many users reported side effects of extended caching warmup times and disc fragmentation problems with their computer afterwards, so I decided to refrain from altering the heap size.

Furthermore, I tried to the best of my ability to keep the searches as computationally inexpensive as possible, however, inevitably there are areas for improvement in terms of time complexity for the simulations. Also, I chose to store my board state representation in a 2d character array, as for the purposes of this report I believe that clearly representing the board state to the reader takes precedence over memory efficiency. However, if this experiment were to be repeated, I would choose a different data structure to represent the array in order to improve the simulation, for example a single digit integer array, as this can still represent the board to a readable degree.

The way in which I found different configurations to get solution depths was to change my start configuration and run the A* search to find out what the solution depth of that particular configuration is. In reality, there are several different configurations for each depth level, and each depth level has a computationally different level of intensity, due to the nature of the board moves. I tried to mitigate this effect on my data by keeping the next start state as close as possible to the preceding one, however, the experimentally sound way to do this would be to calculate every single configuration that has depth level n, and then average the running time for each one of those configurations. Unfortunately, this was too large a task to complete within the parameters of this study.

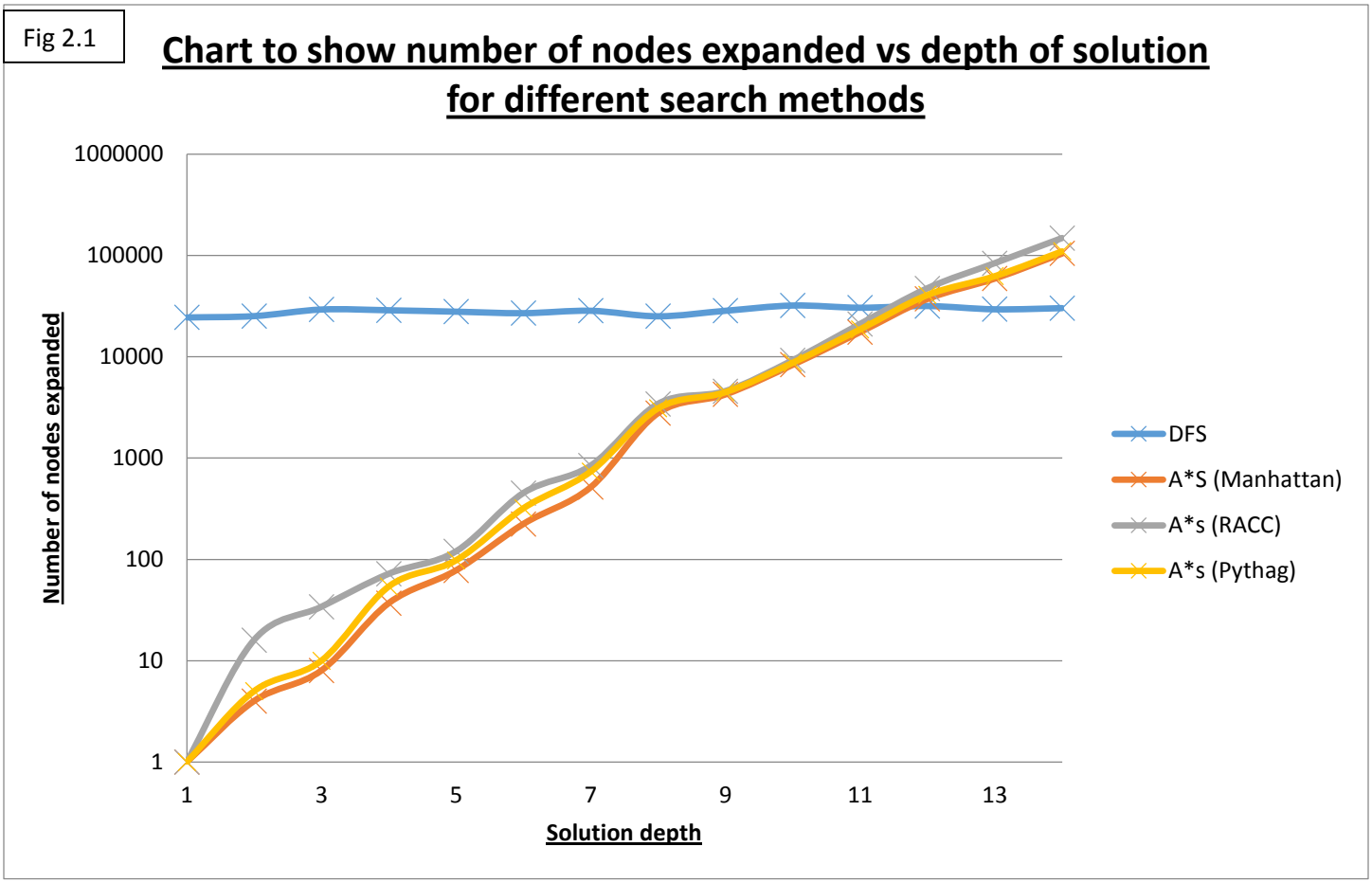
Similarly, as there are four different movements 'up', 'down', 'left', 'right' there are $4! = 24$ different combinations for the order of movements, each permutation of which would affect the order of nodes traversed, and therefore also the results. I tried to mitigate this by including 'Collections.shuffle' to rearrange the order of nodes within the movement method itself. However, with an unrestricted amount of time and computational power, I would instead implement a movement method for each permutation of these movements and run a set of searches for each one.

Extensions

1.) Making the problem scalable for an nxn board as previously explained in 'approach'.
2.) Providing 2 additional heuristics - these were both added to the current depth to obtain the evaluation function (cost).
    - Number of rows and columns in the right place for A,B,C respectively – To implement this I searched the 2d array for the locations of A, B, C. If the row or column was out of place I would add '1' to the heuristic, for a potential of between 0-6 for the heuristic element of the evaluation function. Heuristic method is in the "BoardNode" class, implementation is in the "AStarS2 class"
    - Pythagorean distance using the horizontal and vertical distances to form a triangle and calculate the hypotenuse using $c = \sqrt{a^2 + b^2}$. Heuristic method is in the "BoardNode" class, implementation is in the "AStarS3 class".
3.) Comparing all 3 heuristics to see if an A* that beats depth first search past a depth of 11.62 was produced. For this section of the coursework, I increased the heap space limit using the command listed in my evaluation.

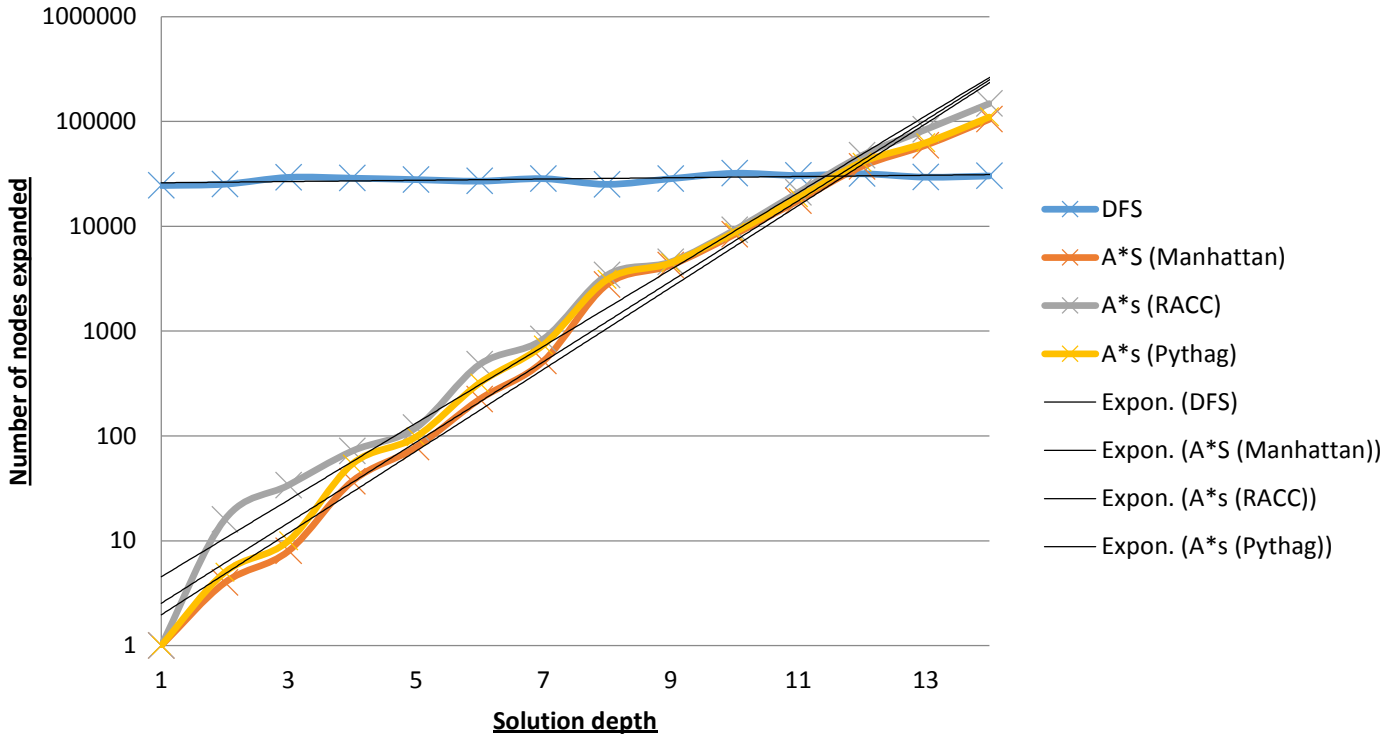
Each search was run 150 times, much like the previous investigation. The search was run up to a solution depth of 14, this was possible due to the increased heap space limit. The results are presented in fig 2.1 below:

Fig 2.1



The results show that the Manhattan distance was the most efficient form of heuristic, followed by Pythagoras, and then Rows and column correctness (RACC). When we analyse the heuristics this becomes evident, as the Pythagoras heuristic essentially takes the information used in the Manhattan distance (horizontal and vertical distance) and simplifies it by calculating the hypotenuse, thereby reducing the amount of information used in the heuristic. This case is similar with the rows and column correctness factor, as the worst possible case for this heuristic is 6, and there is not as much data to differentiate between when compared to the other two search methods. Fig 2.2 with the trendlines is shown below:

Fig 2.2



The graph shows that A* (RACC) intersects the depth first search at 11.35, and A*(pythag) intersects the depth first search at 11.54. So, unfortunately I could not produce a heuristic to beat a solution depth of 11.62.

Bibliography and references

Javarevisited.blogspot.com. (2019). *10 points about Java Heap Space or Java Heap Memory*. [online] Available at: https://javarevisited.blogspot.com/2011/05/java-heap-space-memory-size-jvm.html [Accessed 18 Nov. 2019].

Ai.stanford.edu. (2019). [online] Available at: http://ai.stanford.edu/~latombe/cs121/2011/slides/D-heuristic-search.pdf [Accessed 22 Nov. 2019].

Tutorialspoint.com. (2019). *Data Structure - Depth First Traversal - Tutorialspoint*. [online] Available at: https://www.tutorialspoint.com/data_structures_algorithms/depth_first_traversal.htm [Accessed 15 Nov. 2019].

Medium. (2019). *Solving 8-Puzzle using A* Algorithm.*. [online] Available at: https://blog.goodaudience.com/solving-8-puzzle-using-a-algorithm-7b509c331288 [Accessed 12 Nov. 2019].

Appendices

Manhattan A* search shown with a solution at depth level 4 (Output read top to bottom for each of the 5 blocks):

2.)

3.)

```
[[ , !, A,  ]
[ ,  ,  ,  ]
[ , B,  ,  ]
[ , C,  ,  ]]

Manhattan distance is: 1
Depth is: 1
Cost of next step is: 2
Priority Queue size is: 3
[[ , A, !,  ]
[ ,  ,  ,  ]
[ , B,  ,  ]
[ , C,  ,  ]]

Manhattan distance is: 2
Depth is: 1
Cost of next step is: 3
Priority Queue size is: 5
[[!,  , A,  ]
[ ,  ,  ,  ]
[ , B,  ,  ]
[ , C,  ,  ]]

Manhattan distance is: 2
Depth is: 1
Cost of next step is: 3
Priority Queue size is: 6
[[ ,  , A,  ]
[ , !,  ,  ]
[ , B,  ,  ]
[ , C,  ,  ]]

Manhattan distance is: 1
Depth is: 2
Cost of next step is: 3
Priority Queue size is: 9
[[ , A,  ,  ]
[ ,  , !,  ]
[ , B,  ,  ]
[ , C,  ,  ]]
```

```
Manhattan distance is: 1
Depth is: 2
Cost of next step is: 3
Priority Queue size is: 12
[[ , A,  , !]
[ ,  ,  ,  ]
[ , B,  ,  ]
[ , C,  ,  ]]

Manhattan distance is: 2
Depth is: 2
Cost of next step is: 4
Priority Queue size is: 13
[[ , !, A,  ]
[ ,  ,  ,  ]
[ , B,  ,  ]
[ , C,  ,  ]]

Manhattan distance is: 2
Depth is: 2
Cost of next step is: 4
Priority Queue size is: 15
[[ , !, A,  ]
[ ,  ,  ,  ]
[ , B,  ,  ]
[ , C,  ,  ]]

Manhattan distance is: 2
Depth is: 2
Cost of next step is: 4
Priority Queue size is: 17
[[ ,  , A,  ]
[!,  ,  ,  ]
[ , B,  ,  ]
[ , C,  ,  ]]

Manhattan distance is: 2
Depth is: 2
Cost of next step is: 4
Priority Queue size is: 19
[[ , !, A,  ]
[ ,  ,  ,  ]
[ , B,  ,  ]
[ , C,  ,  ]]
```

1.)

```
Manhattan distance is: 2
Depth is: 2
Cost of next step is: 4
Priority Queue size is: 21
[[ ,  , A,  ]
[ ,  , !,  ]
[ , B,  ,  ]
[ , C,  ,  ]]

Manhattan distance is: 2
Depth is: 2
Cost of next step is: 4
Priority Queue size is: 24
[[ ,  , A,  ]
[!,  ,  ,  ]
[ , B,  ,  ]
[ , C,  ,  ]]

Manhattan distance is: 1
Depth is: 3
Cost of next step is: 4
Priority Queue size is: 26
[[ , A, !,  ]
[ ,  ,  ,  ]
[ , B,  ,  ]
[ , C,  ,  ]]

Manhattan distance is: 1
Depth is: 3
Cost of next step is: 4
Priority Queue size is: 28
[[ , A,  ,  ]
[ ,  ,  ,  ]
[ , B, !,  ]
[ , C,  ,  ]]

Manhattan distance is: 1
Depth is: 3
Cost of next step is: 4
Priority Queue size is: 31
[[ , A,  ,  ]
[ , !,  ,  ]
[ , B,  ,  ]
[ , C,  ,  ]]
```

4.)

```
Manhattan distance is: 1
Depth is: 3
Cost of next step is: 4
Priority Queue size is: 34
[[ , A,  ,  ]
[ ,  ,  , !]
[ , B,  ,  ]
[ , C,  ,  ]]

Manhattan distance is: 1
Depth is: 3
Cost of next step is: 4
Priority Queue size is: 36
[[ , A, !,  ]
[ ,  ,  ,  ]
[ , B,  ,  ]
[ , C,  ,  ]]

Manhattan distance is: 1
Depth is: 3
Cost of next step is: 4
Priority Queue size is: 38
[[ , A,  ,  ]
[ ,  ,  , !]
[ , B,  ,  ]
[ , C,  ,  ]]

Manhattan distance is: 1
Depth is: 3
Cost of next step is: 4
Priority Queue size is: 40
[[ , A, !,  ]
[ ,  ,  ,  ]
[ , B,  ,  ]
[ , C,  ,  ]]

Manhattan distance is: 1
Depth is: 3
Cost of next step is: 4
Priority Queue size is: 42
[[ , A, !,  ]
[ ,  ,  ,  ]
[ , B,  ,  ]
[ , C,  ,  ]]
```

5.)

```
Manhattan distance is: 1
Depth is: 3
Cost of next step is: 4
Priority Queue size is: 44
[[ , A, !,  ]
[ ,  ,  ,  ]
[ , B,  ,  ]
[ , C,  ,  ]]

Manhattan distance is: 1
Depth is: 3
Cost of next step is: 4
Priority Queue size is: 46
[[ ,  , !,  ]
[ ,  , A,  ]
[ , B,  ,  ]
[ , C,  ,  ]]

Manhattan distance is: 0
Depth is: 4
Cost of next step is: 4
Priority Queue size is: 48
[[ , !,  ,  ]
[ , A,  ,  ]
[ , B,  ,  ]
[ , C,  ,  ]]
Depth is: 4
Nodes Generated: 72
```

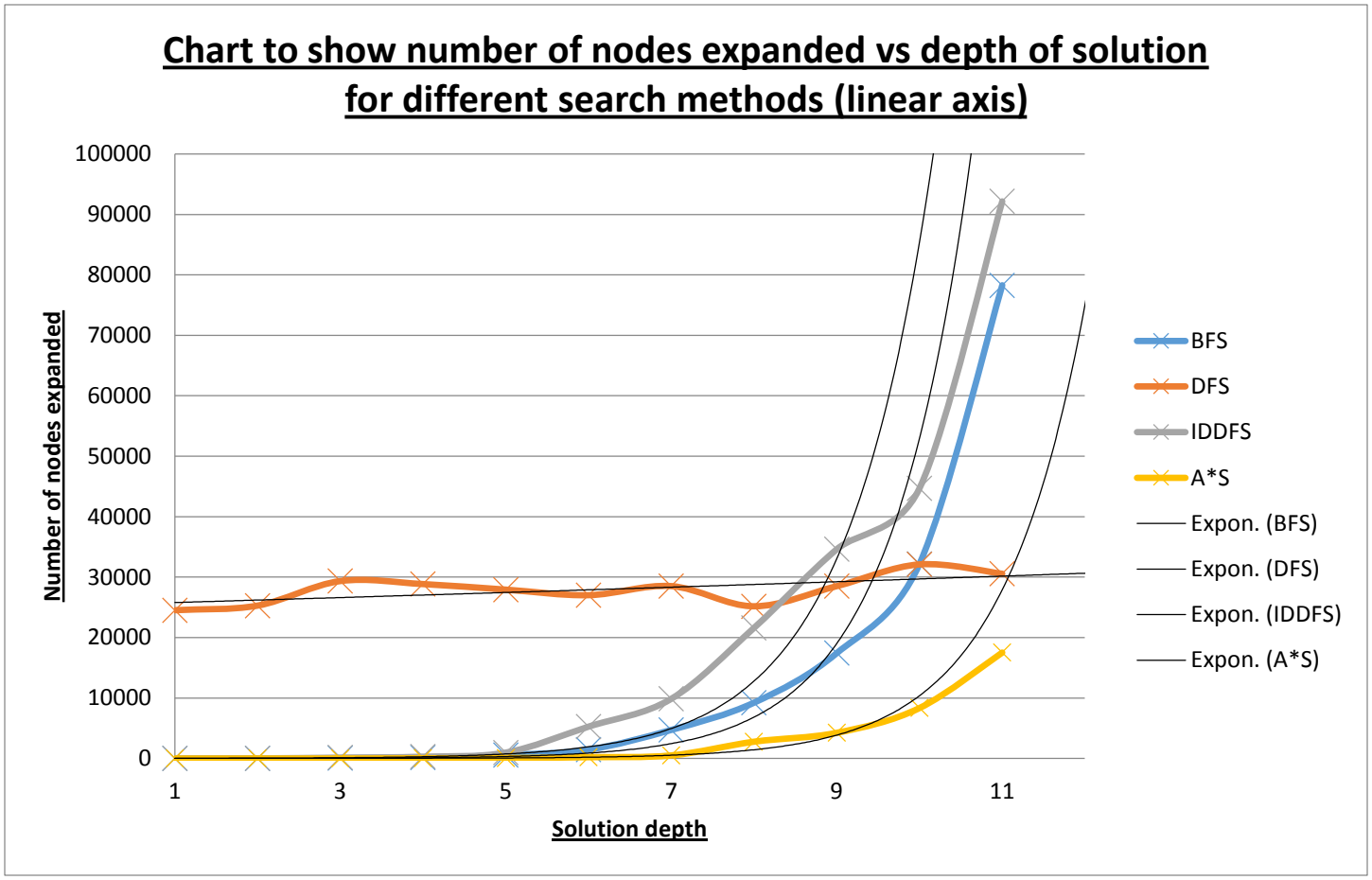


Chart to show number of nodes expanded vs depth of solution for different search methods (linear axis)
Number of nodes expanded
100000
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
1
3
5
7
9
11
Solution depth
BFS
DFS
IDDFS
A*S
Expon. (BFS)
Expon. (DFS)
Expon. (IDDFS)
Expon. (A*S)

BoardNode class:

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;

public class BoardNode {

    public char[][] grid;
    public int rowsAndColumns;
    public intPair agent;
    public BoardNode parent;
    public int depth = 0;
    public int manhattanDistance;
    public int rowsAndColumnsCorrectness;
    public double pythagDistance;



    public BoardNode(int rowsAndColumns) {
        this.rowsAndColumns = rowsAndColumns;
        grid = new char[rowsAndColumns][rowsAndColumns];
        grid[rowsAndColumns - 4][rowsAndColumns - 3] = '!';
        grid[rowsAndColumns - 1][rowsAndColumns - 3] = 'C';
        grid[rowsAndColumns - 2][rowsAndColumns - 3] = 'B';
        grid[rowsAndColumns - 4][rowsAndColumns - 2] = 'A';
        checkGoalState();
        agent = new intPair( row: rowsAndColumns - 4,  column: rowsAndColumns - 3);
        parent = null;
        manhattanDistance();
        rowsAndColumnsCorrectness();
        pythagDistance();
    }

    public BoardNode(BoardNode boardNode){
        this.rowsAndColumns = boardNode.rowsAndColumns;

        this.parent = boardNode;

        grid = new char[rowsAndColumns][rowsAndColumns];

        for(int i = 0; i< boardNode.grid.length; i++){
            for(int j = 0; j< boardNode.grid[0].length; j++){
                grid[i][j]= boardNode.grid[i][j];
            }
        }

       this.agent = new intPair(boardNode.agent.getRow(),boardNode.agent.getColumn());
        manhattanDistance();
        rowsAndColumnsCorrectness();
        pythagDistance();
        this.depth = parent.depth + 1;
    }

    public void printGrid() {
        System.out.println(Arrays.deepToString(this.grid).replace( target: "], ",  replacement: "]\n"));

    }
```

```
    // Method that checks if the board is in a goal state i.e. that A B C are stacked on top
    // of each other with A at the top and C at the bottom
    public boolean checkGoalState() {
        return grid[rowsAndColumns - 3][rowsAndColumns - 3] == 'A'
                && grid[rowsAndColumns - 2][rowsAndColumns - 3] == 'B'
                && grid[rowsAndColumns - 1][rowsAndColumns - 3] == 'C';

    }

    public Boolean movement(char direction) {
        if (direction == ('u') && agent.getRow() != 0) {
            this.grid[agent.getRow()][agent.getColumn()] = this.grid[agent.getRow() - 1][agent.getColumn()];
            this.grid[agent.getRow() - 1][agent.getColumn()] = '!';
            //    this.printGrid();
            agent.setRow(agent.getRow() - 1);
            agent.setColumn(agent.getColumn());
            manhattanDistance();
            rowsAndColumnsCorrectness();
            pythagDistance();
            return true;
        } else if (direction == ('d') && agent.getRow() < (rowsAndColumns - 1)) {
            this.grid[agent.getRow()][agent.getColumn()] = this.grid[agent.getRow() + 1][agent.getColumn()];
            this.grid[agent.getRow() + 1][agent.getColumn()] = '!';
            //      this.printGrid();
            agent.setRow(agent.getRow() + 1);
            agent.setColumn(agent.getColumn());
            manhattanDistance();
            rowsAndColumnsCorrectness();
            pythagDistance();
            return true;
        } else if (direction == ('l') && agent.getColumn() != 0) {
            this.grid[agent.getRow()][agent.getColumn()] = this.grid[agent.getRow()][agent.getColumn() - 1];
            this.grid[agent.getRow()][agent.getColumn() - 1] = '!';
            //     this.printGrid();
            agent.setRow(agent.getRow());
            agent.setColumn(agent.getColumn() - 1);
            manhattanDistance();
            rowsAndColumnsCorrectness();
            pythagDistance();
            return true;

        } else if (direction == ('r') && agent.getColumn() < rowsAndColumns - 1) {
            this.grid[agent.getRow()][agent.getColumn()] = this.grid[agent.getRow()][agent.getColumn() + 1];
            this.grid[agent.getRow()][agent.getColumn() + 1] = '!';
            //    this.printGrid();
            agent.setRow(agent.getRow());
            agent.setColumn(agent.getColumn() + 1);
            manhattanDistance();
            rowsAndColumnsCorrectness();
            pythagDistance();
            return true;
        }

        return false;
    }
```

```
public ArrayList<BoardNode> generateChildren() {
    ArrayList<BoardNode> children = new ArrayList<>();
    BoardNode child1 = new BoardNode(this);
    if (child1.movement( direction: 'u')) {
        children.add(child1);
     //    System.out.println(Arrays.deepToString( children.get(0).

    }

    BoardNode child2 = new BoardNode(this);
    if (child2.movement( direction: 'd')) {
        children.add(child2);
    }

    BoardNode child3 = new BoardNode(this);
    if (child3.movement( direction: 'l')) {
        children.add(child3);

    }

    BoardNode child4 = new BoardNode(this);
    if (child4.movement( direction: 'r')) {
        children.add(child4);
    }

    Collections.shuffle(children);

    return children;
}
```

```
public void manhattanDistance(){
int rowA = 0;
int columnA = 0;
int rowB = 0;
int columnB = 0;
int rowC = 0;
int columnC = 0;
    for(int i = 0; i< grid.length; i++){
        for(int j = 0; j< grid[0].length; j++){
            if (grid[i][j] == 'A'){
               rowA = Math.abs(i - (rowsAndColumns-3));
               columnA = Math.abs(j - (rowsAndColumns-3));

            }

            if (grid[i][j] == 'B'){
                rowB = Math.abs(i - (rowsAndColumns-2));
                columnB = Math.abs(j- (rowsAndColumns-3));
            }

            if (grid[i][j] == 'C'){
                rowC = Math.abs(i-(rowsAndColumns-1));
                columnC = Math.abs(j - (rowsAndColumns-3));
            }

        }
    }

    this.manhattanDistance = (rowA +columnA + rowB + columnB + rowC + columnC);


}
```

```java
public void rowsAndColumnsCorrectness(){
    int rowA = 1;
    int columnA = 1;
    int rowB = 1;
    int columnB = 1;
    int rowC = 1;
    int columnC = 1;
    for(int i = 0; i< grid.length; i++) {
        for (int j = 0; j < grid[0].length; j++) {
            if (grid[i][rowsAndColumns - 3] == 'A') {
                columnA = 0;
            }

            if (grid[rowsAndColumns - 3][j] == 'A') {
                rowA = 0;

            }

            if (grid[i][rowsAndColumns - 3] == 'B') {
                columnB = 0;
            }

            if (grid[rowsAndColumns - 2][j] == 'B') {
                rowB = 0;

            }

            if (grid[i][rowsAndColumns - 3] == 'C') {
                columnC = 0;
            }

            if (grid[rowsAndColumns - 1][j] == 'C') {
                rowC = 0;

            }

            this.rowsAndColumnsCorrectness = (rowA +columnA + rowB + columnB + rowC + columnC);


        }
    }
}
```

```
public void pythagDistance(){
    int rowA = 0;
    int columnA = 0;
    int rowB = 0;
    int columnB = 0;
    int rowC = 0;
    int columnC = 0;
    double pythagA =0;
    double pythagB =0;
    double pythagC = 0;
    double totalA;
    double totalB;
    double totalC;
    double squaredA =0;
    double squaredB =0;
    double squaredC =0;


    for(int i = 0; i< grid.length; i++){
        for(int j = 0; j< grid[0].length; j++){
            if (grid[i][j] == 'A'){
                rowA = (i - (rowsAndColumns-3));
                columnA = (j - (rowsAndColumns-3));
                totalA = (double)(rowA + columnA);
                squaredA = Math.pow(totalA,2);
                pythagA = Math.sqrt(squaredA);
            }

            if (grid[i][j] == 'B'){
                rowB = (i - (rowsAndColumns-2));
                columnB = (j- (rowsAndColumns-3));
                totalB = (double)(rowB + columnB);
                squaredB = Math.pow(totalB,2);
                pythagB = Math.sqrt(squaredB);
            }

            if (grid[i][j] == 'C'){
                rowC = (i-(rowsAndColumns-1));
                columnC = (j - (rowsAndColumns-3));
                totalC = (double)(rowC + columnC);
                squaredC = Math.pow(totalC,2);
                pythagC = Math.sqrt(squaredC);

            }

        }
    }

    this.pythagDistance = (pythagA + pythagB + pythagC);


}
```

```
    public int getCostWManhattan(){
        return (this.depth + this.manhattanDistance);
    }

    public int getCostWRACC(){
        return (this.depth + this.rowsAndColumnsCorrectness);
    }

    public double getCostPythagDistance(){
        return (this.depth + this.pythagDistance);
    }



}

class intPair {
    private int row;
    private int column;

    public intPair(int row, int column) {
        this.row = row;
        this.column = column;
    }

    public int getRow() {
        return row;
    }

    public int getColumn() {
        return column;
    }

    public void setRow(int row) {
        this.row = row;

    }

    public void setColumn(int column) {
        this.column = column;
    }
}
```

Breadth first search class:

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.LinkedList;

public class BFS{

    LinkedList<BoardNode> q;
    int nodesGenerated = 0;

    public BFS(BoardNode start) {
        q = new LinkedList<>();

        System.out.println(Arrays.deepToString(start.grid).replace( target: "], ",  replacement: "]\n"));
        search(start);


    }

    public char[][] search(BoardNode start){

        ArrayList<BoardNode> list = start.generateChildren();

        nodesGenerated = nodesGenerated + list.size();

        if ((list.isEmpty())) {
        } else {
            for (BoardNode grid : list) {
                q.add(grid);
            }
        }

        if (start.checkGoalState()){
            System.out.println("Depth is: " + start.depth);
            System.out.println("Nodes generated: " + nodesGenerated);
            return start.grid;
        }

        for (int i =0; i<list.size();i++ ) {
            System.out.println();
            System.out.println("queue size is " + q.size());
            System.out.println(Arrays.deepToString(q.getFirst().grid).replace( target: "], ",  replacement: "]\n"));
            return search(q.poll());
        }

        return null;
    }
}
```

Depth first search class:

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.LinkedList;

public class DFS{

    LinkedList<BoardNode> q;
    int nodesGenerated = 0;

    public DFS(BoardNode start) {
        q = new LinkedList<>();

        System.out.println(Arrays.deepToString(start.grid).replace( target: "], ",  replacement: "]\n"));
        search(start);


    }

    public char[][] search(BoardNode start){

        ArrayList<BoardNode> list = start.generateChildren();

        nodesGenerated = nodesGenerated + list.size();

        if ((list.isEmpty())) {
        } else {
            for (BoardNode grid : list) {
                q.add(grid);
            }
        }

        if (start.checkGoalState()){
            System.out.println("Depth is: " + start.depth);
            System.out.println("Nodes Generated: " + nodesGenerated);
            return start.grid;
        }

        for (int i =0; i<list.size();i++ ) {
            System.out.println();
            System.out.println("depth is: " + list.get(i).depth);
            System.out.println("'stack' size is " + q.size());
            System.out.println(Arrays.deepToString(q.getLast().grid).replace( target: "], ",  replacement: "]\n"));
            return search(q.pollLast());
        }

        return null;
    }
}
```

Iterative deepening depth first search class:

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.LinkedList;

public class IDDFS{

    LinkedList<BoardNode> q;
    int nodesGenerated = 0;
    int depthLimit = 1;
    BoardNode rootNode;

    public IDDFS(BoardNode start) {
        q = new LinkedList<>();

        rootNode = start;
        System.out.println(Arrays.deepToString(start.grid).replace( target: "], ", replacement: "]\n"));
        search(start);


    }

    public char[][] search(BoardNode start){


            ArrayList<BoardNode> list = start.generateChildren();

        if (!(start.depth >= depthLimit)) {
            nodesGenerated = nodesGenerated + list.size();

            if ((list.isEmpty())) {
            } else {
                for (BoardNode grid : list) {
                    q.add(grid);
                }
            }
        } else {
            depthLimit = depthLimit +1;
            System.out.println("Depth limit increased to: " + depthLimit + " search restarting");
            System.out.println();
            q.clear();
            System.out.println(Arrays.deepToString(rootNode.grid).replace( target: "], ", replacement: "]\n"));
            search(rootNode);
        }

        if (start.checkGoalState()){
            System.out.println("GOAL STATE REACHED!");
            System.out.println("current depth is: " + start.depth);
            System.out.println("depth limit is: " + depthLimit);
            System.out.println("Nodes Generated: " + nodesGenerated);
            return start.grid;
        }

        for (int i =0; i<list.size();i++ ) {
            System.out.println();
            System.out.println("Stack size is: " + q.size());
            System.out.println(Arrays.deepToString(q.getLast().grid).replace( target: "], ", replacement: "]\n"));
            return search(q.pollLast());
        }
```

```
        return null;
    }
}
```

A* Search (Manhattan) class:

```
import java.util.ArrayList;
import java.util.Arrays;

public class AStarS{

    BoardNodePriorityQueue q;
    int nodesGenerated = 0;
    BoardNode lowestCost;

    public AStarS(BoardNode start) {
        q = new BoardNodePriorityQueue();

        lowestCost = start;
        System.out.println(Arrays.deepToString(start.grid).replace( target: "], ", replacement: "]\n"));
        search(start);


    }

    public char[][] search(BoardNode start){

        ArrayList<BoardNode> list = start.generateChildren();

        nodesGenerated = nodesGenerated + list.size();

        if ((list.isEmpty())) {
        } else {
            for (BoardNode grid : list) {
                q.add(grid);
            }
        }

        if (start.checkGoalState()){
            System.out.println("Depth is: " + start.depth);
            System.out.println("Nodes Generated: " + nodesGenerated);
            return start.grid;
        }


        return search(q.pop());



    }
}
```

```java
class BoardNodePriorityQueue{

    ArrayList<BoardNode> queue;

    BoardNodePriorityQueue(){
        queue = new ArrayList<>();
    }

    void add(BoardNode boardNode) {
        queue.add(boardNode);
    }

    BoardNode pop(){

        BoardNode lowest = queue.get(0);
        for (BoardNode x : queue){
            if (x.getCostWManhattan() < lowest.getCostWManhattan()){
                lowest = x;
            }
        }

        int mD = lowest.manhattanDistance;
        int depth = lowest.depth;

        System.out.println();
        System.out.println("Manhattan distance is: " + mD);
        System.out.println("Depth is: " + depth);
        System.out.println("Cost of next step is: " + (mD + depth));
        System.out.println("Priority Queue size is: " + size());
        System.out.println(Arrays.deepToString(lowest.grid).replace( target "], ",  replacement "]\n"));

        queue.remove(lowest);

        return lowest;
    }

    int size(){
       return queue.size();
    }
}
```

A* Search (Rows and columns correctness) class:

```
import java.util.ArrayList;
import java.util.Arrays;

public class AStarS2{

    BoardNodePriorityQueue2 q;
    int nodesGenerated = 0;
    BoardNode lowestCost;

    public AStarS2(BoardNode start) {
        q = new BoardNodePriorityQueue2();

        lowestCost = start;
        System.out.println(Arrays.deepToString(start.grid).replace( target: "], ",  replacement: "]\n"));
        search(start);


    }

    public char[][] search(BoardNode start){

        ArrayList<BoardNode> list = start.generateChildren();

        nodesGenerated = nodesGenerated + list.size();

        if ((list.isEmpty())) {
        } else {
            for (BoardNode grid : list) {
                q.add(grid);
            }
        }

        if (start.checkGoalState()){
            System.out.println("Depth is: " + start.depth);
            System.out.println("Nodes Generated: " + nodesGenerated);
            return start.grid;
        }


        return search(q.pop());



    }
}
```

```java
class BoardNodePriorityQueue2{

    ArrayList<BoardNode> queue;

    BoardNodePriorityQueue2(){
        queue = new ArrayList<>();
    }

    void add(BoardNode boardNode) {
        queue.add(boardNode);
    }

    BoardNode pop(){

        BoardNode lowest = queue.get(0);
        for (BoardNode x : queue){
            if (x.getCostWRACC() < lowest.getCostWRACC()){
                lowest = x;
            }
        }

        int rACC = lowest.rowsAndColumnsCorrectness;
        int depth = lowest.depth;

        System.out.println();
        System.out.println("RACC value is: " + rACC);
        System.out.println("Depth is: " + depth);
        System.out.println("Cost of next step is: " + (rACC + depth));
        System.out.println("Priority Queue size is: " + size());
        System.out.println(Arrays.deepToString(lowest.grid).replace( target: "], ",  replacement: "]\n"));

        queue.remove(lowest);

        return lowest;
    }

    int size(){
        return queue.size();
    }
}
```

A* Search (Pythagoras) class:

```
import java.util.ArrayList;
import java.util.Arrays;

public class AStarS3{

    BoardNodePriorityQueue3 q;
    int nodesGenerated = 0;
    BoardNode lowestCost;

    public AStarS3(BoardNode start) {
        q = new BoardNodePriorityQueue3();

        lowestCost = start;
        System.out.println(Arrays.deepToString(start.grid).replace( target: "], ",  replacement: "]\n"));
        search(start);


    }

    public char[][] search(BoardNode start){

        ArrayList<BoardNode> list = start.generateChildren();

        nodesGenerated = nodesGenerated + list.size();

        if ((list.isEmpty())) {
        } else {
            for (BoardNode grid : list) {
                q.add(grid);
            }
        }

        if (start.checkGoalState()){
            System.out.println("Depth is: " + start.depth);
            System.out.println("Nodes Generated: " + nodesGenerated);
            return start.grid;
        }


        return search(q.pop());



    }
}
```

```java
class BoardNodePriorityQueue3{

    ArrayList<BoardNode> queue;

    BoardNodePriorityQueue3(){
        queue = new ArrayList<>();
    }

    void add(BoardNode boardNode) {
        queue.add(boardNode);
    }

    BoardNode pop(){

        BoardNode lowest = queue.get(0);
        for (BoardNode x : queue){
            if (x.getCostPythagDistance() < lowest.getCostPythagDistance()){
                lowest = x;
            }
        }

        double pyD = lowest.rowsAndColumnsCorrectness;
        double depth = lowest.depth;

        System.out.println();
        System.out.println("PyD value is: " + pyD);
        System.out.println("Depth is: " + depth);
        System.out.println("Cost of next step is: " + (pyD + depth));
        System.out.println("Priority Queue size is: " + size());
        System.out.println(Arrays.deepToString(lowest.grid).replace( target: "], ", replacement: "]\n"));

        queue.remove(lowest);

        return lowest;
    }

    int size(){
        return queue.size();
    }
}
```