Approach

In order to solve the problem of an agent in a world whose aim is to move blocks to obtain a goal configuration, I needed to encode this grid world including the agent, blocks and actions that the agent can take. My modelling was as follows: we have a state which contains all information of a grid world at an instant, the agent can generate new states from a given state by moving.

My approach was to use Java. I encoded a grid data structure using a 2d array of integers. The 0s represented empty tiles, the number 2 represents the agent, numbers 3 4 and 5 represent the distinct blocks.

```
private int agentrow = 3;
private int agentcol = 3;

public int three_row = 3;
public int three_col = 0;
                          private  int[][] start = {{0,0,0,0},
                                                    {0,0,0,0},
                                                    {0,0,0,0},
                                                    {3,4,5,2}};

public int four_row = 3;
public int four_col =0;
public int five_row = 3;
public int five_col = 2;
```

The start state labelled start. Methods for the agents moves were provided inside the grid class (represents state), so this class contains everything that we need to know about the current world, but also can generate new ones.

The other major data structure was the node. As shown below, the node contains a grid structure, depth, the moves the agent took previously and the direct last move and so on. An important detail is the moves String. This contains all of the moves that led up to the node. This gives us the information we need about the move sequence for the agent. It also avoids storing an entire search tree. When generating a new node, this string is assigned as the moves of the previous node in addition to the move undertaken between these nodes.

```
ublic class Node implements Cloneable{

   grid config;
   Node parent = null;
   int pathCost;
   int totalCost;
   int heuristic = 0;

   int depth;
   ArrayList<Node> children;
   String moves = "$";
   String lastMove= "$";
```

The method of interest in the node structure is expand. This generates a list of other nodes containing states that the agent can reach from the present state. The expand method used the encapsulated methods inside the grid world, which represent the agent moving up/down/left/right. I found it useful to begin solving the problem in a top down way. Thinking about what the search methods need (an expand method), what this has to do and so on ; bearing in mind data structures to be stored as well.

In summary, all of the search methods used this expand method to generate new nodes (each containing a new state). Each method removes a node from the structure, runs a goal test and then continues expanding and adding. I deliberately avoided using recursion. For DFS and IDS, a stack structure was used, for BFS a queue and a priority queue structure for A*, In which the priority set by the heuristic function.

During the problem set up, I realised that it is not necessary for the agent to move back into the state it has just come from, as this has already been tested by the goal state function. Therefore in the methods, the agent can only undertake a maximum of 3 moves, not 4. This leads to faster search, because duplicate states are not tested. I realised later that this was a mistake as in a true run of the problem, the agent would not have this insight and simply traverse the world blindly. This decision was not reviewed until later, largely due to the results being of the same pattern. Ultimately I decided to include this feature, because I realised it does not alter the optimal move sequence that the algorithms could find, the overall complexity pattern or the differences in speed.

Most of the bugs came up in the grid world and node structures. There were problems with shared memory, where the new state generated from a move was dependant on the previous state for example, resulting in multiple agents or repeats of other blocks.

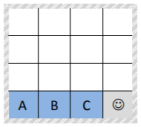
Use

In order to run the search methods correctly, it is key that the data inside the grid class, which stores the positions of the agent and the blocks is correct. If changes are made to the start state or goal state, these have to be done manually.

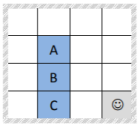
Evidence

Overview

All in all, the search methods all worked managed to find the goal state from the starting configuration (shown below), although DFS occasionally failed as expected.



Start state.



Goal state.

Branching Factor

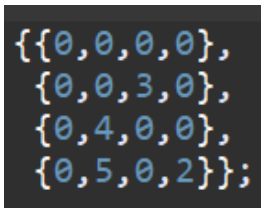
This is useful for predicting results found later.

The branching factor is not constant for this problem; therefore an average is needed. To calculate the average, we take a weighted mean. The branching factor is 1 in the corners, 2 on the other boundary tiles and 3 In the centre tiles. We assume that the agent is equally likely to be in any tile

for purposes of estimation. Therefore, the average branching factor is (8 * 2 + 4 * 3 + 4 * 1)/16 = 2. In actuality, the chances are that the agent is more likely to be in a centre square, because there are more moves going in to them, so the bf is likely to be slightly greater than 2. We can use this to predict an approximate range for nodes expanded.

Test

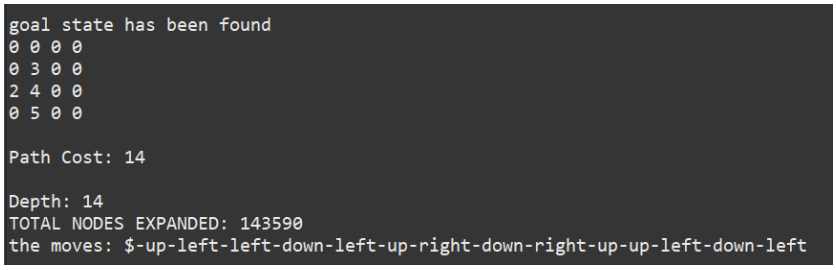
The test case will be the following configuration. In this section the search methods will be run on this case, the results shown. This is shown in addition to the original problem.



This can be solved in 7 moves minimum, by moving (u – u – u – l – l – d – r).

BFS:

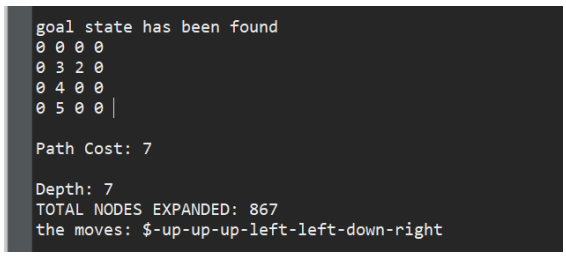
The output:



Estimated output through nodes(2) = 2^(d+1) -1 = 33,000.

nodes(2.3) = 2^(d+1) -1 = 266,635

Test Case(BFS)



Status : passed.

DFS

```
the stack is full, now testing contents
goal state has been found
0 0 0 0
2 3 0 0
0 4 0 0
0 5 0 0

Path Cost: 14825

Depth: 14825
TOTAL NODES EXPANDED: 42142
```

DFS is not an optimal algorithm, without a limit, it will continue moving without backtracking. In this case It found the solution. Interestingly, the stack reached its memory limit during the search and the algorithm threw an error. However, the algorithm caught the error and tested everything remaining in the stack (all of the unexplored expanded nodes). The solution was found in the contents.

Test Case(DFS)

Status: could not find optimum solution but solved. For small depths, it is better to use BFS or another algorithm. DFS will all to often move into the wrong node at some point and not find the optimum solution.

IDS

```
GOAL state has been found
0 0 0 0
0 3 0 0
2 4 0 0
0 5 0 0

the solution found:
0 0 0 0
0 3 0 0
2 4 0 0
0 5 0 0

Path Cost: 14

Depth: 14
TOTAL NODES EXPANDED: 100488
the action sequence: $-up-left-left-down-left-up-right-down-right-up-up-left-down-left
```

IDS managed to find the goal state. It was convincing to me that it is working correctly because the depth and string of moves matches with BFS and A*. This is the optimal solution. The nodes

expanded was different each time due to the random ordering in expansion (same as in DFS). The magnitude of the nodes expanded was correct: O(b^n), as shown by the following calculation :

Nodes(2) = 2^ 14 = 16,384

Nodes(2.3) = 2.3 ^14 = 163,842

The value 100488 falls within this range.

Test Case (IDS)

```
GOAL state has been found
0 0 0 0
0 3 2 0
0 4 0 0
0 5 0 0

the solution found:
0 0 0 0
0 3 2 0
0 4 0 0
0 5 0 0

Path Cost: 7

Depth: 7
TOTAL NODES EXPANDED: 581
the action sequence: $-up-up-up-left-left-down-right
```

Status : Passed.

A*

```
the found configuration:
0 0 0 0
0 3 0 0
2 4 0 0
0 5 0 0

Goal state found at depth: 14
NODES EXPANDED: 2999
the moves: $-up-left-left-down-left-up-right-down-right-up-up-left-down-left
```

The goal state was found as shown above, the depth was the same as in BFS and IDS, as well as the move sequence. This convinced me that the methods are working and that they are finding the optimal solution.

Test Case (A*)

```
the found configuration:
0 0 0 0
0 3 2 0
0 4 0 0
0 5 0 0

Goal state found at depth: 7
NODES EXPANDED: 173
the moves: $-up-up-up-left-left-down-right
```

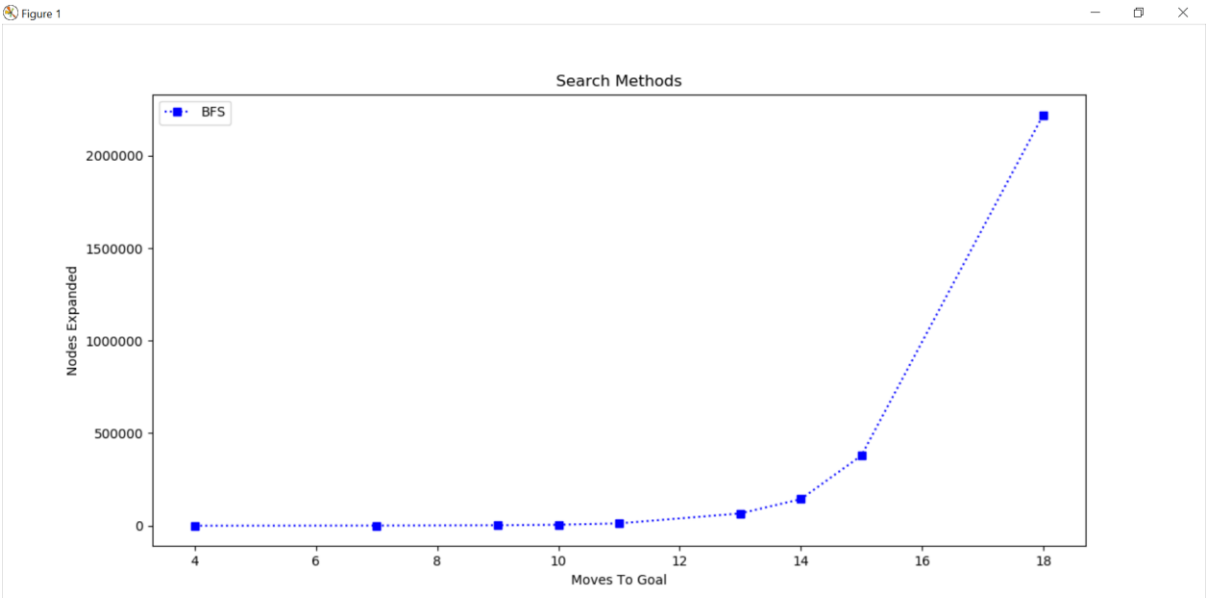
Status: Passed.

Data Analysis

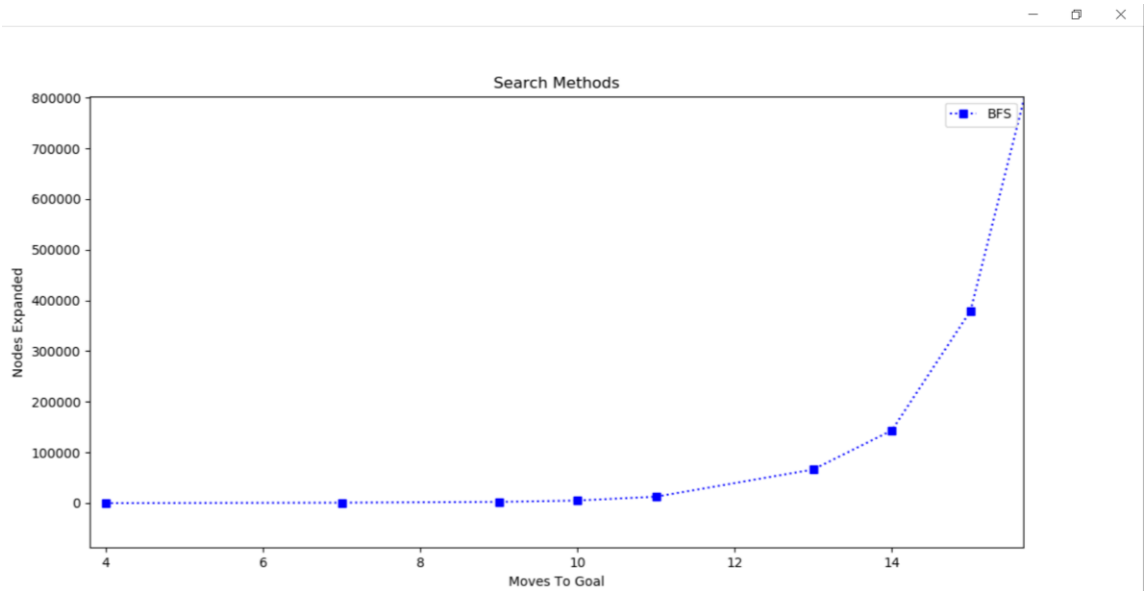
How I Controlled Difficulty

I controlled the problem difficulty by manually altering the start state. I made educated guesses as to what a difficult starting configuration would be i.e. all of the blocks are dispersed vs an easy configuration (two blocks already in correct position/ small Manhattan distance). The actual measure of the problem difficulty was how many moves the agent is required to take in the case of the optimal solution. I thought this intuitive as well as reasonable, because if the agent has to move more, then this suggests that 'more effort' was required, making the problem more difficult.

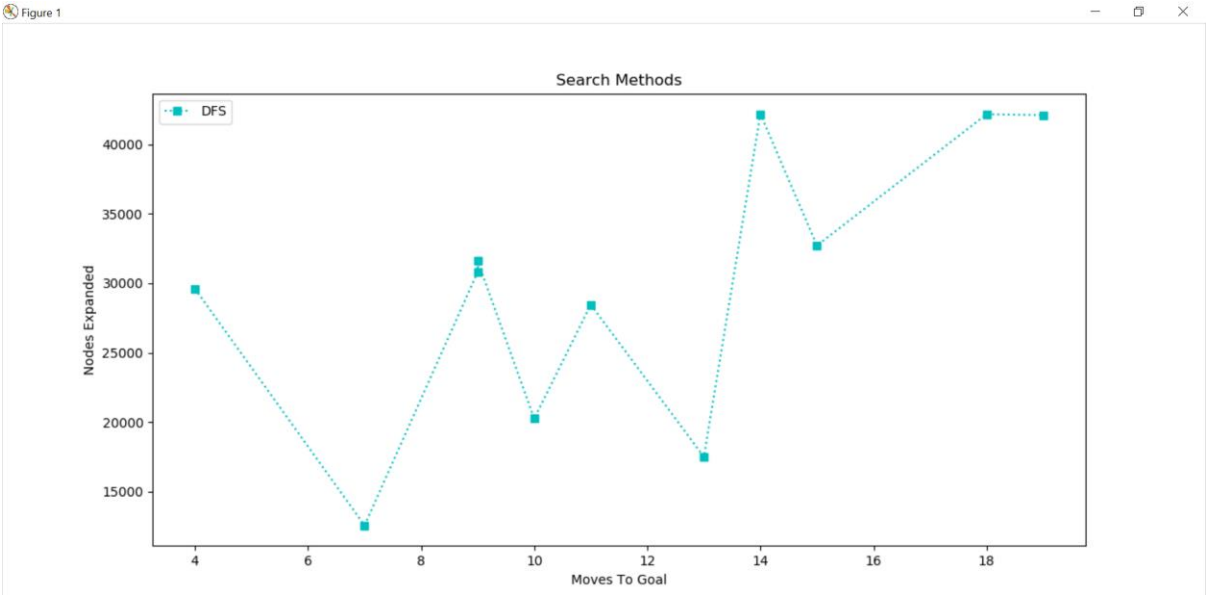
Pattern for **BFS:**



Here we can see an exponential pattern emerging, which is what we would expect for an O(b^d+1) algorithm. Also, the values should be increasing with each step, which is evident with the data. The difference in the numbers of the nodes expanded for various depths makes it difficult to see the structure from moves 4 to 14. We can zoom in on only this part and take a look:



DFS

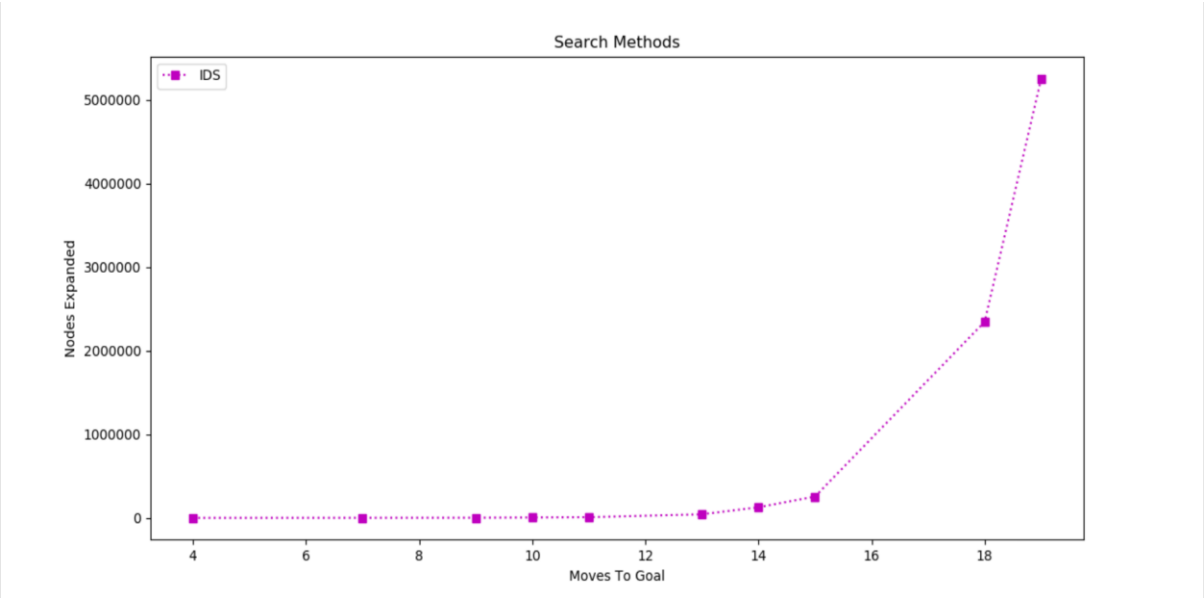


At first sight this is strange, as the complexity of depth first search is expected to be exponential. However, this can be explained. Although configurations whose solutions have great depths require a much larger search space for algorithms like BFS or A*, DFS can overcome this and sometimes get lucky. The depth first search keep making moves, without trying all of the other possibilities until later. Sometimes, the agent will get lucky and randomly find the solution. In other words, because the search is depth first, it can sometimes find deep solutions very quickly. Hence, this is why we can see that solutions of greater depth were sometimes found more quickly than those of shallower depth. Depth first search is not optimal, it will often find a solution that is much deeper than the optimal solution depth.

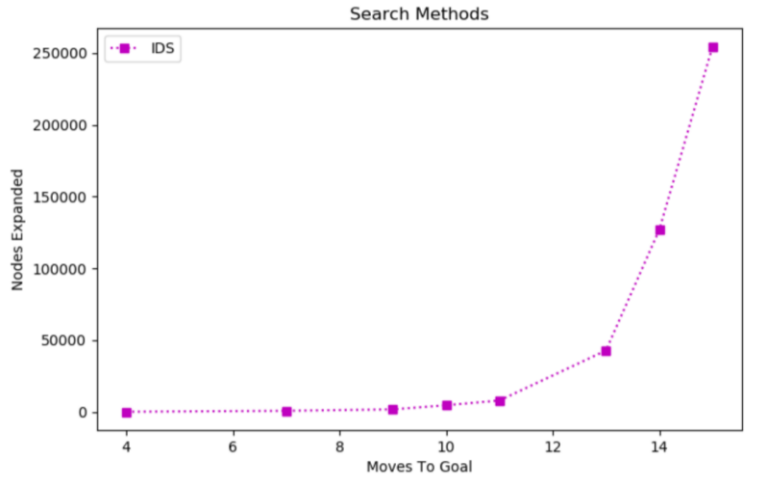
An aspect of the figure that is more predictable is that the general trend seems to be moving upwards. On average, if a solution is deeper in the search tree, it is less likely that the optimal solution will be found, if it exists.

IDS

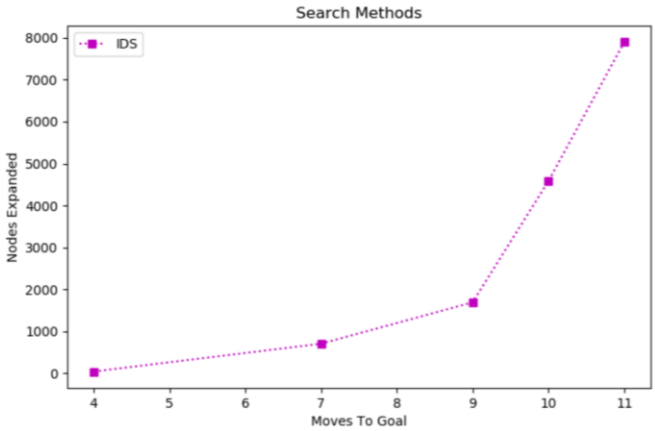
With IDS, we expect an exponential complexity of O(b^d). IDS will find the optimal solution, as it increases its depth iteratively. The main advantage of using IDS over BFS is that it requires less memory. In the results it was possible to find a solution that required 19 moves in the optimum case, but BFS failed.



A view of the pattern for the easier problems :



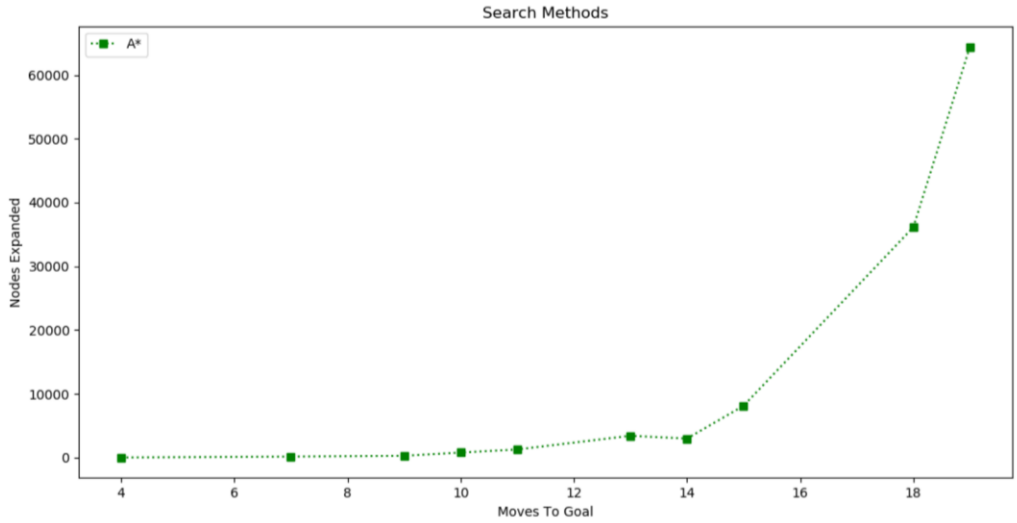
From 4 to 11 moves :



Note that in the figure it shows an equivalent leap between 9-10 and 10-11 which is strange because we are expecting an exponential increase. However, this is not always the case with IDS, which iteratively uses DFS and therefore shares the property of randomly expanding. The ordering of expansion influences the number of nodes generated. For example, if a solution lies in the fringe at depth d, IDS could enter this fringe at any point, so the number of nodes searched in this fringe depends on chance. Overall, the pattern resembles an exponential function.

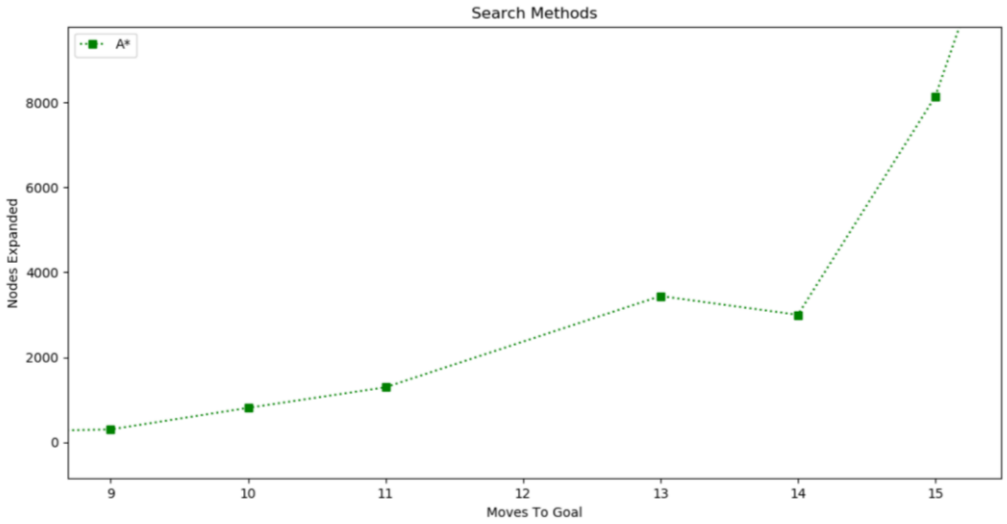
A*

The figure for A* search:



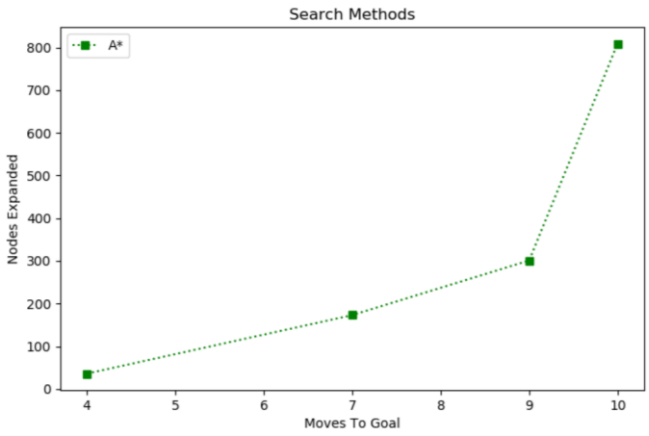
At first glance the pattern appears to be exponential, although focusing is needed for the earlier points. This is what we expect for an A* algorithm of O(b^d).

Zoom:



Something to note here is the kink. The algorithm actually found a solution for 14 moves away quicker than that of 13. This is consistent with how A* operates. The speed at which it finds a solution depends on the heuristic, as well as the depth of the search required. Sometimes the heuristic will narrow in very quickly on the solution. An element of luck is needed for this to happen.

Further simplification of data set:



Extras and Limitations

An additional feature that I added was that it was not possible for an agent to move into a tile it had previously moved from. My reasoning behind this was that if the agent undertook this operation, it would end up in a state already visited previously. If this state did not pass the goal test, then it is obsolete. Furthermore, it has obviously already been expanded. The problem with this is that it is adding to the original problem space. A problem with this is that the search is now not completely blind, which is not how the algorithms are supposed to be defined.

In my evidence section, I discussed the problem of estimating a branching factor. The estimate I calculated directly was 2, but I reasoned that it is probably slightly higher than this. As a result, I didn't have an exact branching factor to work with, making my predictions for nodes expanded potentially unreliable. That being said, I did decide to work with 2 and 2.3, the nodes expanded fell within the range of predictions using these figures.

Source Code

Grid

```
public class grid {

        //Agent : 2

        //BLOCKS: 3, 4 , 5

        //NOTE : the start and goal configurations have to manually be matched up to the data!
        //start
        private int agentrow = 3;
        private int agentcol = 3;

        public int three_row = 3;
        public int three_col = 0;
                        private  int[][] start = {{0,0,0,0},
                                                  {0,0,0,0},
                                                  {0,0,0,0},
                                                {3,4,5,2}};

        public int four_row = 3;
        public int four_col =1;
        public int five_row = 3;
        public int five_col = 2;
```

```
	//goal : NOTE IF THIS IS CHANGED, THE MANHATTEN DISTANCE NUMBERS HAVE TO BE ALTERED.
	public int three_row_goal =1;
	public int three_col_goal = 1;
	public int four_row_goal = 2;
	public int four_col_goal = 1;
	public int five_row_goal = 3;
	public int five_col_goal = 1;
	public static final int[][] goal =  {{0,0,0,0},
	                                     {0,3,0,0},
	                                     {0,4,0,0},
	                                     {0,5,0,2}};

	int[][] config;
	public int getrow() {
		return this.agentrow;
	}

	public int getcol() {
		return this.agentcol;
	}

	public int[][] getconfig() {
		return this.config;
	}

	public grid(int col, int row) {
```

```
		this.config = start;

		this.setAgentPos(col, row);



}

public boolean goalthree() {
		if (this.three_col == this.three_col_goal && this.three_row == this.three_row_goal) {
			return true;
		}
		else {
			return false;
		}
}
public boolean goalfour() {
		if (this.four_col == this.four_col_goal && this.four_row == this.four_row_goal) {
			return true;
		}
		else {
			return false;
		}
}


public boolean goalfive() {
		if (this.five_col == this.five_col_goal && this.five_row == this.five_row_goal) {
			return true;
		}
		else {
```

```
                return false;
            }
        }

        //when we are creating grid, we need to state exactly the positions of all of the blocks and the agent.
        public grid(int[][] config, int agentcol, int agentrow, int threecol, int threerow, int fourcol, int fourrow, int fivecol, int fiverow) {
            this.config = config;
            this.agentcol = agentcol;
            this.agentrow= agentrow;
            this.three_col = threecol;
            this.three_row = threerow;
            this.four_col = fourcol;
            this.four_row = fourrow;
            this.five_col = fivecol;
            this.five_row = fiverow;
        }

        public grid(int[][] config, int row, int col) {
            this.agentcol = col;
            this.agentrow = row;
            this.setConfig(config);
            this.setAgentPos(col, row);
        }

        public void setConfig(int[][] config) {
            this.config = config;
        }
```

```
//in coordinates where we want the agent to be -> Out: An altered configuration and agent
coordinates.
public  void setAgentPos(int col, int row) {

        int temp;
        if (!(row == this.agentrow && col == this.agentcol)) {
                //we reset

                //what is currently at the new position
                temp = this.getconfig()[row][col];

                if (temp== 3) {
                        three_row = getrow();
                        three_col = getcol();
                }
                if (temp== 4) {
                        four_row = getrow();
                        four_col = getcol();
                }
                if (temp== 5) {
                        five_row = getrow();
                        five_col = getcol();
                }



                this.getconfig()[getrow()][getcol()] = temp;

                this.getconfig()[row][col] = 2;

                this.agentcol = col;
                this.agentrow = row;
```

```
}
}
```

Node

```
public class Node implements Cloneable{

	grid config;
	Node parent = null;
	int pathCost;
	int totalCost;
	int heuristic = 0;

	int depth;
	ArrayList<Node> children;
	String moves = "$";
	String lastMove= "$";

	public Node(grid state,int cost, int depth) {

		this.config = state;
		this.children = new ArrayList<Node>();

		this.pathCost = cost;
		this.depth = depth;
		this.totalCost = pathCost + heuristic;

		//add empty stack symbol so search doesn't fail immediately.

	}

	public void addMove(String s) {
```

```
            moves = moves.concat("-").concat(s);

        }

        public Node(grid state,int cost,int depth,Node parent,String move, String moves) {

            this.config = state;
            this.children = new ArrayList<Node>();

            this.pathCost = cost;
            this.depth = depth;
            this.parent = parent;
            this.setMoves(moves);
            this.addMove(move);
            this.lastMove = move;


        }

public Node(grid state,int cost,int depth,Node parent,String move, String moves,boolean val) {

            this.config = state;
            this.children = new ArrayList<Node>();

            this.pathCost = cost;
            this.depth = depth;
            this.parent = parent;
            this.setMoves(moves);
            this.addMove(move);
            this.lastMove = move;
```

```
		//the heuristic. we would also have to call a heuristic function to set the heuristic.
		this.heuristic = this.config.manhattenDist();
		this.totalCost = pathCost + heuristic;
	}


	public void setMoves(String moves) {
		this.moves = moves;
	}

	public void setStartNode() {
		depth = 0;
		pathCost = 0;
		parent = null;

	}

	public void setParent(Node n) {
		this.parent = n;

	}
	public String getMoves() {
		return this.moves;
	}

	//we want to take a node n, return a list of new expanded nodes, without interfering with the state n.
	public ArrayList<Node> expand(Node n, boolean val) {
		ArrayList<Node> nodelist = new ArrayList<Node>();
		grid current = n.config;
		Node newnode;
```

```
		try {
			//don't move into the place where we have just come from.

			//A* search expansion.
			if (val == true) {
				if (n.lastMove !=  "left") {
					newnode = new Node(current.mr(),n.pathCost+1,n.depth +1
,n,"right",this.getMoves(),val);

					nodelist.add(newnode);
					}
			}
			//uninformed search expansion.
			else {
			if (n.lastMove !=  "left") {
			newnode = new Node(current.mr(),n.pathCost+1,n.depth +1
,n,"right",this.getMoves());

			nodelist.add(newnode);
			}
			}
		}

		catch (IndexOutOfBoundsException e) {
			//we are out of bounds.
		}

		try {
			if (val == true) {
```

```
                    if (n.lastMove != "right") {
                        newnode = new Node(current.ml(),n.pathCost+1,n.depth +1
,n,"left",this.getMoves(),true);
                        nodelist.add(newnode);
                    }
                    }

                    else {
                    if (n.lastMove != "right") {
                    newnode = new Node(current.ml(),n.pathCost+1,n.depth +1
,n,"left",this.getMoves());
                    nodelist.add(newnode);
                    }}
                    }

                catch (IndexOutOfBoundsException e) {
                    //we are out of bounds.
                }




                try {

                    if (val == true) {
                        if (n.lastMove != "down") {
                            newnode = new Node(current.mu(),n.pathCost+1,n.depth
+1,n,"up",this.getMoves(),true);
                            nodelist.add(newnode);
                            }
                    }
                    else {
```

```
                    if (n.lastMove != "down") {
                    newnode = new Node(current.mu(),n.pathCost+1,n.depth
+1,n,"up",this.getMoves());
                    nodelist.add(newnode);
                    }}




                }
                catch (IndexOutOfBoundsException e) {
                    //we are out of bounds.
                }




                try {


                    if (val == true) {
                        if (n.lastMove != "up") {
                            newnode = new Node(current.md(),n.pathCost+1,n.depth
+1 ,n,"down",this.getMoves(),true);
                            nodelist.add(newnode);
                            }
                    }


                    else {
                    if (n.lastMove != "up") {
                    newnode = new Node(current.md(),n.pathCost+1,n.depth +1
,n,"down",this.getMoves());
                    nodelist.add(newnode);
```

```
                }
                }
        }
        catch (IndexOutOfBoundsException e) {
                //we are out of bounds.
        }

        return nodelist;
}

public int getPathCost() {
        return this.pathCost;
}

public int getDepth() {
        return this.depth;
}
```

//this method takes a list and randomly shuffles the contents. This will be used for search methods like depth first where we will need to access the list in random order.

```
public ArrayList<Node> nodeshuffle(ArrayList<Node> array) {
        Random rand = new Random();
        int x1,x2;

        if (array.size() == 1) {
```

```
                return array;
        }
        if (array.size() == 2) {
                try {
                        Collections.swap(array, 0, 1);
                }
                catch (IndexOutOfBoundsException e) {
                        e.printStackTrace();
                }
        }
        //do length + 2 times
        for (int i = 0; i < array.size() + 2; i++) {
                x1 = rand.nextInt(array.size() -1);
                x2 = rand.nextInt(array.size()-1);
                try {
                Collections.swap(array,x1,x2);
                }
                catch (IndexOutOfBoundsException e) {
                        e.printStackTrace();
                }
        }
        return array;
}



public static void main(String[] args) {
        //convincing evidence that the node expansion is working well.
        //the nodes are not allowed to move to their parent node. The parent is not
considered a neighbour,
```

```
        grid g = new grid(2,2);
        g.printgrid();

        /*Node n = new Node(g,0,0);
        for (Node node: n.expand(n,true)) {

                System.out.println("cost of node: " + node.totalCost);
                for (Node n1 : node.expand(node,true)) {
                        System.out.println("cost of node: " + node.totalCost);
                        n1.config.printgrid();
                }
        }
        */

        Node n = new Node(g,0,2);

        for (Node node : n.expand(n, false)) {
                System.out.println(node.lastMove);

        }

        System.out.println("the shuffled array");
        for (Node node : n.nodeshuffle(n.expand(n, false))) {
                System.out.println(node.lastMove);

        }

}}
```

Node Heuristic

```
public class heuristic implements Comparator<Node>{



	//minimum cost has highest priority.
	@Override
	public int compare(Node o1, Node o2) {
		// TODO Auto-generated method stub
		return ( (o1.totalCost - o2.totalCost));
	}

}
```

Search

```
public class Search {

	static int nodesExpanded;

	public static void main(String[] args) {

		//new grid with initial agent position.

		grid startgrid = new grid(3,3);
		System.out.print("the starting grid config: " );
		startgrid.printgrid();

		Node startnode = new Node(startgrid,0,0);
		try {
		IDS(startnode);
		}
```

```
		catch(Exception e) {
			System.out.println("failure");
		}
}



public static int IDS(Node start) throws Exception {
		//iterate between depth 0 and 20.

		int depth = 0;
		nodesExpanded = 0;
		ArrayList<Node> currentnodes;
		Node current;


		for (int limit = 1; limit < 20; limit++) {
			depth = limit;


		System.out.println("new limit: " + limit);
		Stack<Node> stack = new Stack<Node>();
		stack.push(start);



		while(!stack.isEmpty()) {

			current = stack.pop();
			//System.out.println("popped from stack: ");
			//current.config.printgrid();
```

```
                        //goal test
                        if (current.config.goalthree() && current.config.goalfour() &&
current.config.goalfive()) {
                                System.out.println("GOAL state has been found");
                                current.config.printgrid();
                                System.out.println("the solution found: ");
                                current.config.printgrid();
                                System.out.println("Path Cost: " + current.getPathCost());
                                System.out.println("");
                                System.out.println("Depth: " + current.getDepth());
                                System.out.println("TOTAL NODES EXPANDED: " + nodesExpanded);
                                System.out.println("the action sequence: " + current.getMoves());
                                return nodesExpanded;
                        }


                        if (current.getDepth() <= limit) {
                                System.out.println("the configuration to be expanded at depth: " +
current.getDepth());
                                current.config.printgrid();

                                currentnodes = current.expand(current, false);
                                Collections.shuffle(currentnodes);

                                for (Node n : currentnodes)  {

                                        System.out.println("node depth: " + n.depth);
                                        n.config.printgrid();

                                        stack.push(n);
                                        nodesExpanded++;
```

```
                        }

                }
                //otherwise we continue.

        }
        }

        System.out.println("we could not find a solution for limit : " + depth);
        throw new Exception();

}


public static int DLS(Node start, int limit) throws Exception {

        //initialise
        Node current;
        Stack<Node> stack = new Stack<Node>();
        stack.push(start);
        nodesExpanded = 0;
        ArrayList<Node> expandnodes;


        while(!stack.isEmpty()) {

                current = stack.pop();
                //System.out.println("popped from stack: ");
                //current.config.printgrid();
```

```
                    //goal test
                    if (current.config.goalthree() && current.config.goalfour() &&
current.config.goalfive()) {
                            System.out.println("GOAL state has been found");
                            System.out.println("the solution found: ");
                            current.config.printgrid();
                            System.out.println("Path Cost: " + current.getPathCost());
                            System.out.println("");
                            System.out.println("Depth: " + current.getDepth());
                            System.out.println("TOTAL NODES EXPANDED: " + nodesExpanded);
                            System.out.println("MOVES: " + current.moves);
                            return nodesExpanded;
                    }



                            //expanding if we are not beyond limit.
                    try {
                    if (current.getDepth() <= limit) {
                            System.out.println("the node to be expanded: ");
                            current.config.printgrid();

                            expandnodes  = current.expand(current, false);
                            Collections.shuffle(expandnodes);
                            for (Node n : expandnodes) {
                                    System.out.println("expand nodes: ");
                                    n.config.printgrid();

                                    stack.push(n);
                                    nodesExpanded++;
                            }
```

```
                }

                else {
                        System.out.println("we are beyond the limit for this node");
                }
                }

                catch (OutOfMemoryError e) {
                        System.out.println("the stack is full, now testing contents");
                        //now just pop everything in the stack and test for goal
                        while (!stack.isEmpty()) {
                                current = stack.pop();
                                if (current.config.goalthree() && current.config.goalfour()
&& current.config.goalfive()) {
                                        System.out.println("goal state has been found");
                                        current.config.printgrid();
                                        System.out.println("Path Cost: " +
current.getPathCost());
                                        System.out.println("");
                                        System.out.println("Depth: " + current.getDepth());
                                        System.out.println("TOTAL NODES EXPANDED: " +
nodesExpanded);
                                        return nodesExpanded;
                                }

                        }




                }
                }
```

```
            throw new Exception("Could not find solution for limit: " + limit);


        }


        public  static int DFS(Node start) throws Exception {
            //initialise
            nodesExpanded= 0;
            Node current;

            Stack<Node> stack = new Stack<Node>();
            stack.push(start);
            //a structure to store the expanded nodes.
            ArrayList<Node> expandnodes;

            while(!stack.isEmpty()) {
                current = stack.pop();

                System.out.println("                    \n)"
                        + "Stack size: " + stack.size() + "   \n " +
                        "                              ");

                //goal test
                if (current.config.goalthree() && current.config.goalfour() &&
current.config.goalfive()) {
```

```
                System.out.println("goal state has been found");
                current.config.printgrid();
                System.out.println("Path Cost: " + current.getPathCost());
                System.out.println("");
                System.out.println("Depth: " + current.getDepth());
                System.out.println("TOTAL NODES EXPANDED: " + nodesExpanded);
                System.out.println("the moves: " + current.moves);
                return nodesExpanded;
        }

        //catch memory exception then just empty the stack.
        try {
        System.out.println("the node to be expanded: ");
        current.config.printgrid();

        expandnodes  = current.expand(current, false);
        //ensure that the nodes are in random order -> call a function which
essentially swaps entries randomly.
        Collections.shuffle(expandnodes);
        for (Node n : expandnodes) {
                System.out.println("expand nodes: ");
                n.config.printgrid();

                stack.push(n);
                nodesExpanded++;
        }

        }
        catch (OutOfMemoryError e) {
                System.out.println("the stack is full, now testing contents");
                //now just pop everything in the stack and test for goal
```

```
                while (!stack.isEmpty()) {
                        current = stack.pop();
                        if (current.config.goalthree() && current.config.goalfour()
&& current.config.goalfive()) {
                                System.out.println("goal state has been found");
                                current.config.printgrid();
                                System.out.println("Path Cost: " +
current.getPathCost());
                                System.out.println("");
                                System.out.println("Depth: " + current.getDepth());
                                System.out.println("TOTAL NODES EXPANDED: " +
nodesExpanded);
                                return nodesExpanded;
                        }

                }
                System.out.println("No Solution Found");
                throw new Exception();
            }
            //expand the node, add to the queue.

        }
        System.out.println("Search Failed");
        throw new Exception();
    }
```

```
public  static Node BFS(Node start) throws Exception {
        //initialise
        Node current;
        nodesExpanded = 0;

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

        queue.add(start);

        while(!queue.isEmpty()) {

            current = queue.remove(0);
            System.out.println("                    \n)"
                    + "Queue size: " + queue.size() + "   \n " +
                    "                          ");



            //goal test
            if (current.config.goalthree() && current.config.goalfour() &&
current.config.goalfive()) {
                System.out.println("goal state has been found");
                current.config.printgrid();
                System.out.println("Path Cost: " + current.getPathCost());
                System.out.println("");
                System.out.println("Depth: " + current.getDepth());
                System.out.println("TOTAL NODES EXPANDED: " + nodesExpanded);
                System.out.println("the moves: " + current.moves);
                return current;
            }
            System.out.println("the state to be expanded: ");
            current.config.printgrid();
```

```
                for (Node n : current.expand(current,false)) {
                        if (nodesExpanded < 100000) {

                        System.out.println("expanded: ");
                        n.config.printgrid();
                        }
                        System.out.println("DEPTH: " + current.getDepth());



                        queue.add(n);


                        nodesExpanded++;


                }
                //expand the node, add to the queue.

        }
        System.out.println("Search Failed");
        throw new Exception();
}




public static int astar(Node start) throws Exception {
        //make a priority queue with a comparator.
        PriorityQueue<Node> nodeQueue = new PriorityQueue<Node>(11,new heuristic());
        nodeQueue.add(start);
        Node currentbest;
        nodesExpanded = 0;
```

```
		while(!nodeQueue.isEmpty()) {
			//remove the top of queue.
			currentbest = nodeQueue.remove();
			System.out.println("the state to be expanded; ");
			currentbest.config.printgrid();

			//do a goal test
			if (currentbest.heuristic == 0 && currentbest.depth!= 0) {
				System.out.println("the found configuration: ");
				currentbest.config.printgrid();
				System.out.println("Goal state found at depth: " +
currentbest.getDepth());
				System.out.println("NODES EXPANDED: " + nodesExpanded);

				System.out.println("the moves: " + currentbest.moves);
				return nodesExpanded;

			}

			//expand and add to queue.
			for (Node n : currentbest.expand(currentbest,true)) {
				nodesExpanded++;
				nodeQueue.add(n);

				System.out.println("adding config to queue: ");
				n.config.printgrid();

				System.out.println("current highest priority config with totalcost  " +
nodeQueue.peek().totalCost);
				nodeQueue.peek().config.printgrid();
			}
```

```
                }
                throw new Exception("failure");
        }

}
```