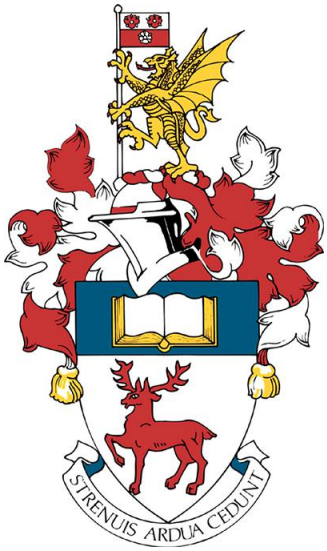
University of Southampton



Intelligent Systems

COMP2208

Blockwords

Author:
Martin Kanev

November, 2019

Contents

1 Approach

For this assignment I chose to use Java, since I most familiar with it. I have investigated how each of the 4 search methods reaches the goal. I have kept track of the node expansion (count), depth and memory used. I have tried to make my code as fast possible, using as little memory as possible. However, my code is not perfect and not fully optimised, but it works. In this report I use BFS for breadth first search, DFS for depth first search, IDS for iterative deepening search and A* for A star heuristic search method.

1.1 Setup

I have decided to store the states of the puzzle in **Nodes**. Each **Node** has a 2D array grid containing the state of the puzzle. It also has a list of children and a parent reference. For the A* search method the **Node** also utilises stored value for the priority, Manhattan distance, depth and for the extended case, a blocked character. In all the searches I have used some collections or structures to keep track of the depth and node count. If these were removed, the code would be even faster and use less memory.

1.2 BFS

For BFS I have decided to use a queue, since it has a very good time complexity of insertion and deletion $O(1)$. First a node is enqueued and then dequeued, expanded and each child tested to be the goal state, then all the children are enqueued. The root is checked at the beginning for being the goal state. Some numbers manipulation is used to keep track of the depth and count. After every loop I delete whatever it is not needed (Node, children list), since there is a parent reference and the children are already in the queue. This gave me some satisfactory space complexity (considering the type of search);

1.3 DFS

For this method I used a stack, which has very good insertion and deletion $O(1)$. I push a node(the root in the beginning) on the stack, pop it, check if it is the goal, if yes, stop everything and give the solution, if no, then expand the node and put its children on the stack and repeat. The children nodes are shuffled and randomly put on the stack. The depth and node count were easily tracked, since it just goes down. However I have also deleted the current nodes and their children list for the same reasons given in BFS section.

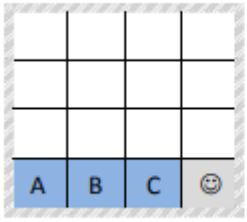
1.4 IDS

I have used the same idea as in DFS. Each time I go one level deeper. When I reach the limit I check the Nodes at that limit and then go up to the deepest unchecked node (closest relative - using the family notation). Going up I delete every reference and nodes and children I can, to the point of preserving the idea of the search method. This deleting has given me some great space complexity. To keep track of the depth and count I have used some numbers manipulations and a hashmap, storing how many nodes each level has, instead of Nodes and their depth. In this way I have less pairs in the hashmap, which uses less memory.

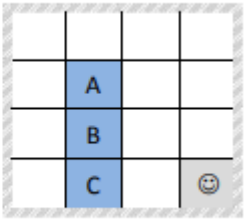
1.5 A*

I use some ideas of the BFS, but I used a priority queue and a comparator. The priority is calculated based on the Manhattan distance and the current depth. I have used scalar multiplications for both of them making the Manhattan distance more important, but at the same time making sure that the depth doesn't fall behind. All these conditions used for the "priority" have given a very good time complexity, however they have used some memory. However, due to the fact that the nodes expanded were so few compared to uninformed search method, the memory was not an issue in our case. A hashmap was used to keep track of the depth and nodes count.

2 Evidence



Start state.



Goal state.

For the puzzle I have decided to represent the agent with '$' , all the others, except the letters, are taken as empty tiles. This can be seen in Appendix 6.1.9. When searching for the solution the methods print to the output the current depth and the number of nodes. When producing the answer it prints each **Grid** and the number above the grid is the depth of that state.

2.1 BFS

The code for BFS can be seen in Appendix 6.1.2. The queue that is used provides the First-in-First-out type of implementation, which is what is needed. BFS produces an optimal solution (path cost is 1) that can be seen in Appendix 6.2.1. Also in Appendix 6.2.3 can be seen the BFS debug- with all the depths and nodes for that depth, until it reaches a solution . I believe this is correct, however the final number of nodes expanded can vary. depending on the way the expansion method works(right, left, up, down- in my case). Furthermore, if there was another optimal solution, changing the sequence of the expansion, can give another optimal solution.

2.2 DFS

The code for DFS can be seen in Appendix 6.1.3. I have used a stack providing Last-in-Fist-out, which is what is needed. The output of the method varies almost every time, because of the way a node's children are put on the stack (random). However the important thing is that it reaches to a solution. I have provided a solution in Appendix 6.2.2, however because it is very long I have given only the beginning and the end. A DFS debug for a different solution can be seen in Appendix 6.2.4.

2.3 IDS

The code for IDS can be seen in Appendix 6.1.4. The solution varies almost every time, for the same reason as DFS. Sometimes it finds a solution with less expanded nodes than BFS, other times it can be worse. The output is an optimal solution (path cost = 1) , however, again if there were another optimal solution, it could reach to it. The debug for the output is in Appendix 6.2.5.

2.4 A*

The code for A* can be seen in Appendix 6.1.5. The method produces an optimal solution. The debug of the method can be seen in Appendix 6.2.6. An interesting thing here to see is that it grows 5 levels and then it drops, then another 5 and drops again. This is due to the multiplication factor.

3 Scalability study

I have controlled the problem difficulty by first getting the optimal solution and then using its states for goal states with shallower depth. This was mainly done, in order to see the complexity of DFS. A graph and a chart for all the methods can be seen in Appendix 6.3.1. And I have use the log function to improve the readability of the graph in Appendix 6.3.2.

3.1 BFS

For BFS I have used the goal state of the problem and examine the previous levels. As we can see for the graph the time complexity is exponential. We can also see that the branching factor is approximately 3. In addition, I have experimented with some obstacles and the problem gets, so big that I run out of memory. This can be overcome in several ways, for example I allowed the heap to use more memory, but the point is that exponential is not good for this problem.

3.2 DFS

DFS for our problem, goes down until it finds a solution. To get the numbers in the chart I have used a helper class in Appendix 6.1.7. The code was run 11 times and I have used the median value. Yet again as expected, the output varies drastically, depending on luck (randomness). An important thing to note here is that, because I have used different goal states for the different levels, and I have taken the median and because of the problem, in the graph DFS looks better than BFS for higher values of depth.

3.3 IDS

The complexity is the same as BFS for all the levels before the final one. This means that the search works correctly. The complexity is still exponential, but if we are lucky we can find the goal faster than BFS.

3.4 A*

The worst case for A* is exponential, but because of the good heuristic (Manhattan distance + depth) many nodes are "pruned" away and the complexity is much better. For every level the complexity is better than for any of the other methods. I have experimented with different multiplication factors and I have found out that 30 times Manhattan distance(+-4) and 11 times depth (it is good around 11 and 15) gives an optimal solution and good time complexity.

3.5 Conclusions

Out of the four search methods, A* heuristic search performs drastically better and it gives an optimal solution. It proves that just by adding a little bit of thought into the search method it pays very well. Out of the uninformed search methods IDS combines the good qualities of BFS and the good of DFS, but still for more difficult problems they all will suffer terribly. Although DFS produces sometimes solutions, with not so bad complexity compared to BFS, it is still a bad choice if an optimal solution is needed.

4 Extras

4.1 Space Complexity

I have managed to track the space complexity of all the 4 methods. In this report the space complexity is calculated using these method calls (Appendix 6.1.8) and my finding were summarised in a graph and a chart in Appendix 6.4.4 and Appendix 6.4.5. The values in the graphs represent memory in KB(RAM). Again I have used the log function to improve readability of the graph.

4.1.1 BFS

As we can see in the chart the rate at which BFS space complexity grows is exponential and it is what it is expected, since it holds all it nodes in memory.

4.1.2 DFS

For this method the memory used correlates directly with the number for node expanded.

4.1.3 IDS

Here the memory used is impressive, since it has a depth limit it gives me the chance of deleting **Nodes** entirely, with all its references. So at depth 14 with average branching factor of 3 it gives $14 * 3 = 42$, we also have to take into account that I delete some objects when I check them, but since there is a parent reference that is no problem. So, to check if the findings are correct, we can use these 42 nodes and compare this with the time complexity of BFS (between level 3 and 4) and go to the memory used by BFS (719KB- 730KB) for that depth and we can see that indeed 721KB is possible. The memory used for IDS still grow slightly, mainly because of the Hashmap that was used for storing the depth and corresponding number of nodes for that depth.

4.1.4 A*

The memory usage of A* is also very good. It is so low, because of the number of nodes generated. And again if we compare A* solution is at 522 nodes this corresponds to a depth between 5 and 6 of BFS of 271- 873 nodes, respectively. Now we compare the memory used, 857KB for A* and between 768KB- 891KB for BFS, which means the memory recording were correct.

4.2 Tree to a graph

I have experimented by making it a graph search problem. It was relatively easy. All that was needed was a condition that the new nodes are not in the current data structure and that they have not been traversed. The time complexity improved drastically, especially for uninformed search methods.

4.3 Obstacles

I have made the problem, a little bit more different by introducing obstacles (non-moving tiles in the puzzle). These fields are marked with 'x' and example can be seen in Appendix 6.4.1. I have changed the **Node** expansion method, so that when the agent look at an "obstacle" it cannot slide there. I have used A* to find the goal stat. The debug and solution can be seen in Appendix 6.4.2 and Appendix 6.4.3. An interesting thing to notice here is that by introducing "obstacles" at certain positions, the complexity gets better, because it cuts some parts of the searching tree.

5 Limitations

- Instead of having a 2D array for the grid, I could have chosen a 1D array, which for bigger puzzles would have had some benefits, but I have kept the 2D array, because it made it easier when I calculated the priority of the A* search method.
- For the A* method, I could have added into an account the number of misplaced tiles, but I don't think this would have changed the things drastically for our puzzle. However for some different setting it should have an impact, but this needs further investigation.
- For the methods that find the children of a Node, I could have used a method call, and maybe this would have made it more human readable.
- If I had more time I could have investigated puzzles that are non-squares and different grid sizes 2X2, 3X3, 5X5.

6 Appendix

6.1 Code

6.1.1 Node.java

```
package blockwords;

import java.util.ArrayList;

public class Node {

    public List<Node> children = new ArrayList<Node>();
    public Node parent;
    public int col = 4;
    char[][] grid = new char[col][col];
    public int priority;
    public int depth;
    public char blocked = 'x';
    public int manhattan;

    public Node(char[][] begin) {
        copyNode(begin, grid);
    }

    public void ExpandNode() {
        int agentRow = 0;
        int agentCol = 0;
        for (int i = 0; i < col; i++) {
            for (int j = 0; j < col; j++) {
                if (grid[i][j] == '$') {
                    agentRow = i;
                    agentCol = j;
                }
            }
        }
        moveRight(grid, agentRow, agentCol);
        moveLeft(grid, agentRow, agentCol);
        moveUp(grid, agentRow, agentCol);
        moveDown(grid, agentRow, agentCol);
    }

    public void moveRight(char[][] some, int agentRow, int agentCol) {


        if (agentCol < col - 1 && some[agentRow][agentCol+1] != blocked) {

            char[][] ch = new char[col][col];
            copyNode(some, ch);

            char temp = ch[agentRow][agentCol + 1];
            ch[agentRow][agentCol + 1] = ch[agentRow][agentCol];
            ch[agentRow][agentCol] = temp;

            Node child = new Node(ch);
            children.add(child);
            child.parent = this;
        }
    }

    public void moveLeft(char[][] some, int agentRow, int agentCol) {
```

```
        char temp = ch[agentRow][agentCol - 1];
        ch[agentRow][agentCol - 1] = ch[agentRow][agentCol];
        ch[agentRow][agentCol] = temp;

        Node child = new Node(ch);
        children.add(child);
        child.parent = this;
    }
}

public void moveUp(char[][] some, int agentRow, int agentCol) {
    if (agentRow > 0 &&  some[agentRow-1][agentCol] != blocked) {
        char[][] ch = new char[col][col];
        copyNode(some, ch);

        char temp = ch[agentRow - 1][agentCol];
        ch[agentRow - 1][agentCol] = ch[agentRow][agentCol];
        ch[agentRow][agentCol] = temp;

        Node child = new Node(ch);
        children.add(child);
        child.parent = this;
    }
}

public void moveDown(char[][] some, int agentRow, int agentCol) {
    if (agentRow < col - 1 &&  some[agentRow+1][agentCol] != blocked) {
        char[][] ch = new char[col][col];
        copyNode(some, ch);

        char temp = ch[agentRow + 1][agentCol];
        ch[agentRow + 1][agentCol] = ch[agentRow][agentCol];
        ch[agentRow][agentCol] = temp;

        Node child = new Node(ch);
        children.add(child);
        child.parent = this;
    }
}


public void calcPriority() {
    int rowDiff = 0;
    int colDiff = 0;
    int val = 0;
    for (int i = 0; i < col; i++) {
        for (int j = 0; j < col; j++) {
            if(this.grid[i][j] == 'a') {
                rowDiff = Math.abs(i - 1);
                colDiff = Math.abs(j - 1);
                val = val + rowDiff + colDiff;
            }
            if(this.grid[i][j] == 'b') {
                rowDiff = Math.abs(i - 2);
                colDiff = Math.abs(j - 1);
                val = val + rowDiff + colDiff;
            }
```

```
            if(this.grid[i][j] == 'c') {
                rowDiff = Math.abs(i - 3);
                colDiff = Math.abs(j - 1);
                val = val + rowDiff + colDiff;

            }

        }

    }
    manhattan = 30*val + (depth*11);
}
public int getPriority() {
    return manhattan;
}

public void printGrid() {
    for (int i = 0; i < col; i++) {
        for (int j = 0; j < col; j++) {
            System.out.print(grid[i][j] + " ");
        }
        System.out.println();

    }
    System.out.println();
}

public boolean puzzleEquality(char[][] x) {
    boolean equal = true;
    for (int i = 0; i < col; i++) {
        for (int j = 0; j < col; j++) {
            if (grid[i][j] != x[i][j]) {
                equal = false;
            }
        }
    }
    return equal;
}

public void copyNode(char[][] x, char[][] y) {
    for (int i = 0; i < x.length; i++) {
        for (int j = 0; j < x.length; j++) {
            y[i][j] = x[i][j];
        }
    }
}

public boolean isGoal() {
    boolean finish = false;
    if (this.grid[1][1] == 'a' && this.grid[2][1] == 'b'
            && this.grid[3][1] == 'c') {
        finish = true;
    }
    return finish;
}

public void setDepth(int depth) {
    this.depth = depth;
}
```

6.1.2 BFS.java

```
public List<Node> BFS(Node root) {
    Queue<Node> queue = new LinkedList<>();
    int depth = 0;
    int count = 1;
    int dcount = 1;
    int dhelper = 0;
    boolean solFound = false;
    queue.add(root);
    if (root.isGoal()) {
        System.out.println("Goal depth " + depth + ", Nodes expanded " + count);
        findPath(solution, root);
        solFound = true;
    }

    here: while (queue.size() > 0 && solFound != true) {
        Node current = queue.poll();
        int count2 = 0;

        current.ExpandNode();

        for (int i = 0; i < current.children.size(); i++) {
            count++;

            Node child = current.children.get(i);

            if (child.isGoal()) {
                System.out.println("Goal is found at depth " + (depth + 1));
                System.out.println("Nodes expanded " + count);
                findPath(solution, child);
                solFound = true;
                break here;
            }

            queue.add(child);
            count2++;

        }
        current.children = null;
        current = null;

        dhelper = dhelper + count2;
        count2 = 0;
        dcount--;
        if (dcount == 0) {
            depth++;
            dcount = dhelper;
            dhelper = 0;
            System.out.println("Depth " + depth + ", " + "Nodes: " + count);
        }
    }
    return solution;
}
```

6.1.3 DFS.java

```
public List<Node> DFS(Node root) {
    Stack<Node> s = new Stack<>();
    int depth = 0;
    int count = 1;
    boolean dbool = false;
    s.push(root);

    here: while (!s.empty() && solFound != true) {

        Node current = s.pop();
        dbool = true;

        if (current.isGoal()) {
            System.out.println(depth + 1);
            runtime.gc();
            // Calculate the used memory
            long memory = runtime.totalMemory() - runtime.freeMemory();
            System.out.println("Memory " + (memory / kb));
            findPath(solution, current);
            solFound = true;
            break here;
        }

        current.ExpandNode();

        Collections.shuffle(current.children);

        for (int i = 0; i < current.children.size(); i++) {
            count++;

            Node child = current.children.get(i);

            if (dbool) {
                depth++;
                dbool = false;
            }
            s.push(child);
        }
        current.children = null;
        current = null;

    }
    return solution;
}
```

6.1.4 IDS.java

```
public List<Node> IDS(Node root) {

    List<Node> ids = new ArrayList<>();
    int maxDepth = 0;
    if (maxDepth == 0) {
        if (root.isGoal()) {
            System.out.println("The goal is the root");
            findPath(solution, root);
        }
        maxDepth = 1;
    }

    while (ids.isEmpty()) {
        ids = DLS(root, maxDepth);
        maxDepth++;

    }
    return ids;
}

public List<Node> DLS(Node root, int maxDepthh) {
    s = new Stack<>();
    HashMap<Integer, Integer> map = new HashMap<>();
    int depth = 0;
    int count = 0;
    int totalCount = 1;
    int maxDepth = maxDepthh;
    boolean delete = false;
    int toDelete = 0;

    s.push(root);
    map.put(0, 1);

    here: while (!s.empty() && solFound != true) {
        Node current = s.pop();
        toDelete = 0;
        delete = false;
        count = 0;

        if (depth < maxDepth) {

            if (current.isGoal()) {
                System.out.println("Goal depth " + depth +
                        ", Nodes expanded " + totalCount);
                solFound = true;
                findPath(solution, current);
                break here;
            }

            current.ExpandNode();
            Collections.shuffle(current.children);

            for (int i = 0; i < current.children.size(); i++) {

                Node child = current.children.get(i);
                s.push(child);
                count++;
                totalCount++;
            }
```

```
        Node child = current.children.get(i);
        s.push(child);
        count++;
        totalCount++;
    }

    current.children = new ArrayList<>();

    if (depth != 0) {
        map.put(depth, map.get(depth) - 1);
        if (count == 0) {
            while (map.get(depth) == 0) {
                depth = depth - 1;
                toDelete++;
                delete = true;
            }
            if (delete = true) {
                for (int i = 0; i < toDelete; i++) {
                    Node n = current.parent;
                    current.parent = null;
                    current = n;
                    n = null;
                }
            }
        }
    }

    if (count != 0) {
        depth = depth + 1;
        map.put(depth, count);
    }

    current = null;


} else {

    if (current.isGoal()) {
        System.out.println("Goal depth " + depth +
                ", Nodes expanded " + totalCount);
        solFound = true;
        findPath(solution, current);
        break here;
    }
    map.put(depth, map.get(depth) - 1);

    while (map.get(depth) == 0) {
        depth = depth - 1;
        toDelete++;
        delete = true;
    }
```

```
            while (map.get(depth) == 0) {
                depth = depth - 1;
                toDelete++;
                delete = true;
            }
            if (delete = true) {
                for (int i = 0; i < toDelete; i++) {
                    Node n = current.parent;
                    current.parent = null;
                    current = n;
                    n = null;
                }
            }

            current = null;

        }

    }
    root = null;
    map = null;
    s = null;
    System.gc();
    System.out.println("Depth " + maxDepth +
            ", Nodes: " + totalCount);
    return solution;
}
```

6.1.5 A*

```
public List<Node> AS(Node root) {
    Comparator<Node> comparator = new MyComparator();
    PriorityQueue<Node> pq = new PriorityQueue<>(comparator);
    HashMap<Node, Integer> map = new HashMap<>();
    HashMap<Node, Integer> nodeCount = new HashMap<>();
    int depth = 0;
    int count = 1;
    boolean dbool = true;

    pq.add(root);

    map.put(root, 0);
    boolean solFound = false;

    while (pq.size() > 0 && solFound != true) {
        Node current = pq.poll();
        depth = map.get(current);

        traversed.add(current);
        dbool = true;

        if (current.isGoal()) {
            System.out.println("Goal depth " + depth +
                    ", Nodes expanded " + count);
            solFound = true;
            findPath(solution, current);
            break;
        }

        current.ExpandNode();

        for (int i = 0; i < current.children.size(); i++) {
            count++;
            if (dbool) {
                depth++;
                dbool = false;
            }

            Node child = current.children.get(i);

            nodeCount.put(child, depth);

            child.setDepth(depth);
            child.calcPriority();
            map.put(child, depth);
            pq.add(child);

        }
```

```
        current.children = null;
        current = null;

    }
    int[] z = new int[14];
    for (Entry<Node, Integer> entry : nodeCount.entrySet()) {
        switch(entry.getValue()) {
        case 1:
            z[0] = z[0]+1;
            break;
        case 2:
            z[1] = z[1]+1;
            break;
        case 3:
            z[2] = z[2]+1;
            break;
        case 4:
            z[3] = z[3]+1;
            break;
        case 5:
            z[4] = z[4]+1;
            break;
        case 6:
            z[5] = z[5]+1;
            break;
        case 7:
            z[6] = z[6]+1;
            break;
        case 8:
            z[7] = z[7]+1;
            break;
        case 9:
            z[8] = z[8]+1;
            break;
        case 10:
            z[9] = z[9]+1;
            break;
        case 11:
            z[10] = z[10]+1;
            break;
        case 12:
            z[11] = z[11]+1;
            break;
        case 13:
            z[12] = z[12]+1;
            break;
        case 14:
            z[13] = z[13]+1;
            break;
        }
    }
    int i = 0;
    for(int x :z) {
        i++;
        System.out.println("Depth " + i +
                ", Nodes: " + x);
    }
    System.out.println("Total nodes " + count);
    return solution;
```

6.1.6 Helping methods

```
class MyComparator implements Comparator<Node> {

    @Override
    public int compare(Node o1, Node o2) {
        priority1 = o1.getPriority();
        priority2 = o2.getPriority();
        if (priority1 > priority2) {
            return 1;
        } else if (priority1 < priority2) {
            return -1;
        } else {
            return 0;
        }
    }
}

public void findPath(List<Node> list, Node a) {
    Node node = a;
    list.add(node);
    while (node.parent != null) {
        node = node.parent;
        list.add(node);
    }
}
```

6.1.7 RunTime.java

```java
package blockwords;

import java.util.Arrays;

public class RunTime {

    static double[] medianTime = new double[11];

    public static void main(String[] args) {
            for (int i = 0; i < 11; i++) {
                long time_prev = System.nanoTime();
                Main.main(null);
                double time = (System.nanoTime() - time_prev) / 1000000000.0;
                medianTime[i] = time;
            }
            System.out.println(median(medianTime));

    }

    public static double median(double[] a) {
        Arrays.sort(a);
        return a[a.length / 2];
    }
}

```

6.1.8 Memory code

```java
runtime.gc();
long memory = runtime.totalMemory() - runtime.freeMemory();
System.out.println("Memory " + (memory / kb));
```

6.1.9 Main code(No obstacles)

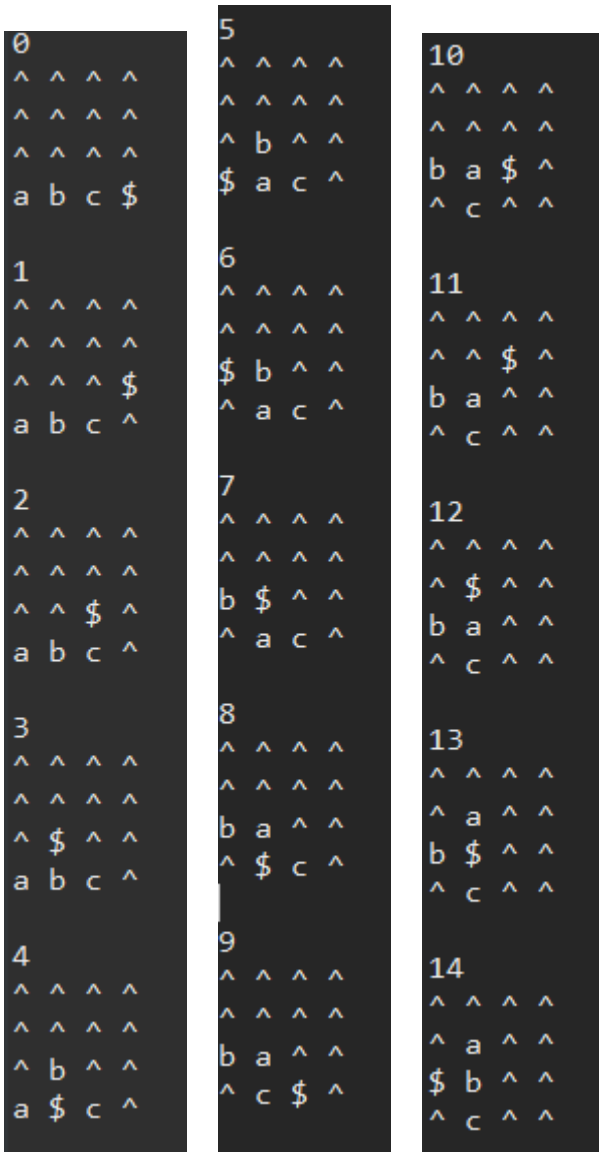
```java
package blockwords;

import java.util.Collections;

public class Main {

    public static void main(String[] args) {
        char[][] startState = {
                {'^','^','^','^'},
                {'^','^','^','^'},
                {'^','^','^','^'},
                {'a','b','c','$'}
        };
        Node root = new Node(startState);

        Search search = new Search();
        List<Node> sol = search.AS(root);

        if(sol.size() > 0) {
            Collections.reverse(sol);
            for(int i = 0; i < sol.size(); i++) {
                System.out.println(i);
                sol.get(i).printGrid();
            }
        } else {
            System.out.println("No solution found");
        }
    }
}
```

6.1.10 Main code(Obstacles)

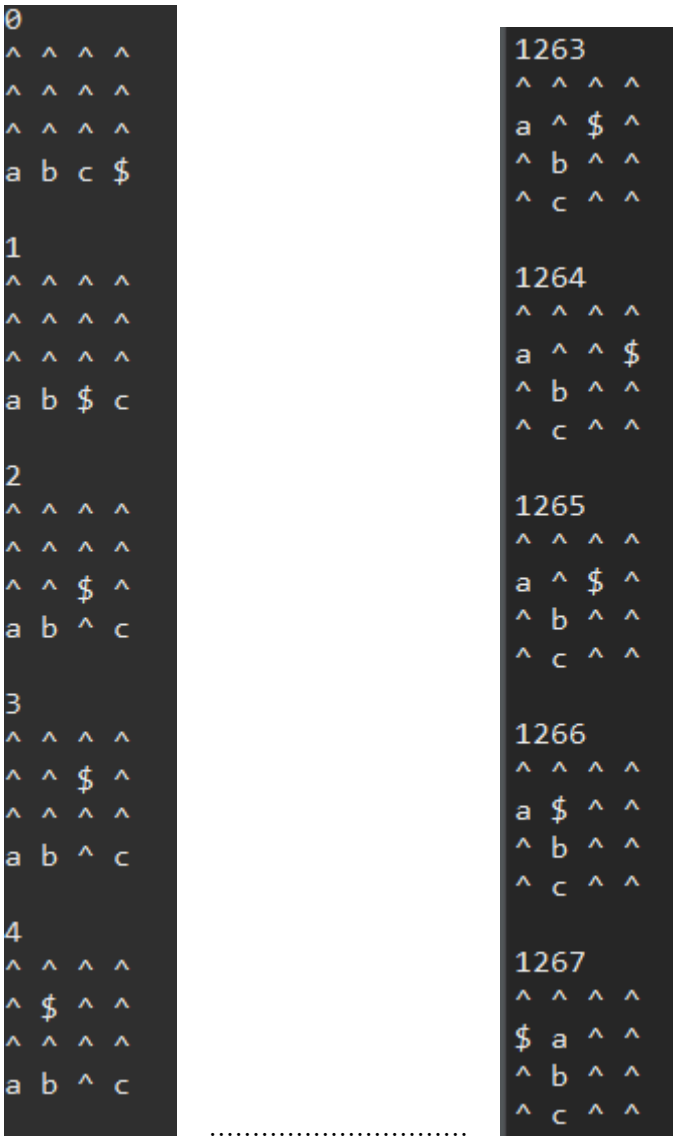
```java
package blockwords;

import java.util.Collections;

public class Main {

    public static void main(String[] args) {
        char[][] startState = {
                {'^','^','^','^'},
                {'^','^','x','^'},
                {'^','^','x','^'},
                {'a','b','c','$'}
        };
        Node root = new Node(startState);

        Search search = new Search();
        List<Node> sol = search.AS(root);

        if(sol.size() > 0) {
            Collections.reverse(sol);
            for(int i = 0; i < sol.size(); i++) {
                System.out.println(i);
                sol.get(i).printGrid();
            }
        } else {
            System.out.println("No solution found");
        }
    }
}
```

6.2 Output

6.2.1 Optimal solution



6.2.2 DFS -solution



6.2.3 BFS- debug

```
Depth 1, Nodes: 3
Depth 2, Nodes: 9
Depth 3, Nodes: 27
Depth 4, Nodes: 85
Depth 5, Nodes: 271
Depth 6, Nodes: 873
Depth 7, Nodes: 2819
Depth 8, Nodes: 9117
Depth 9, Nodes: 29495
Depth 10, Nodes: 95441
Depth 11, Nodes: 308843
Depth 12, Nodes: 999429
Depth 13, Nodes: 3234207
Goal is found at depth 14
Nodes expanded 7694972
```

6.2.4 DFS - debug

```
Depth 46131, Nodes: 146303
Depth 46132, Nodes: 146307
Depth 46133, Nodes: 146310
Depth 46134, Nodes: 146314
Depth 46135, Nodes: 146318
Goal depth 46135, Nodes expanded 146318
```

6.2.5 IDS - debug

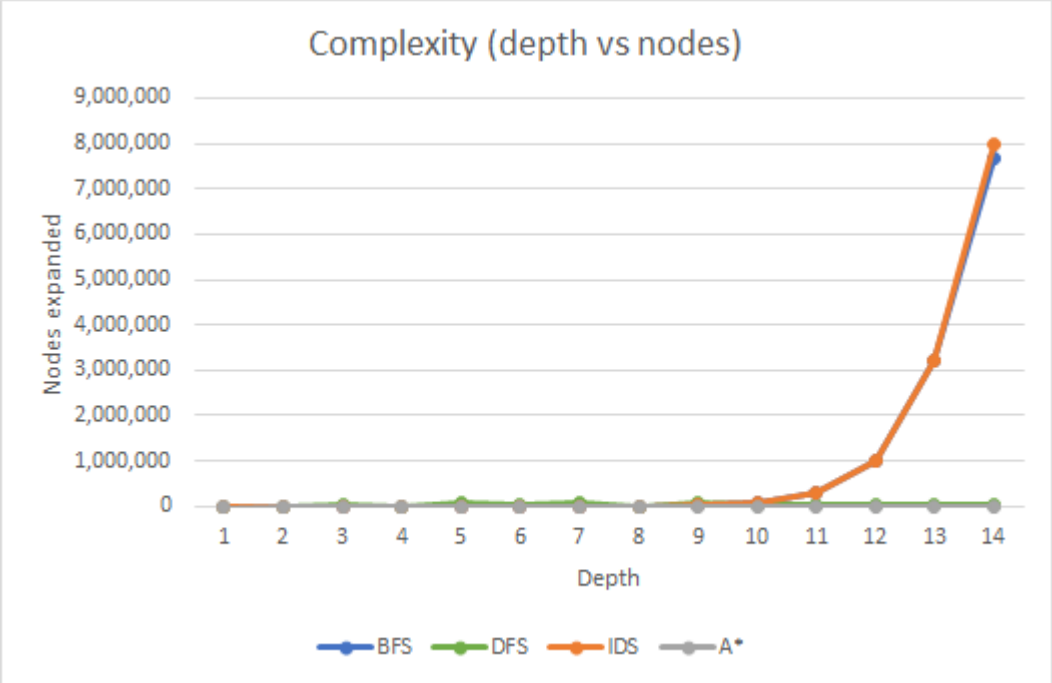
```
Depth 1, Nodes: 3
Depth 2, Nodes: 9
Depth 3, Nodes: 27
Depth 4, Nodes: 85
Depth 5, Nodes: 271
Depth 6, Nodes: 873
Depth 7, Nodes: 2819
Depth 8, Nodes: 9117
Depth 9, Nodes: 29495
Depth 10, Nodes: 95441
Depth 11, Nodes: 308843
Depth 12, Nodes: 999429
Depth 13, Nodes: 3234207
Goal depth 14, Nodes expanded 5960458
```

6.2.6 A* - debug

```
Goal depth 14, Nodes expanded 522
Depth 1, Nodes: 2
Depth 2, Nodes: 6
Depth 3, Nodes: 11
Depth 4, Nodes: 26
Depth 5, Nodes: 84
Depth 6, Nodes: 7
Depth 7, Nodes: 15
Depth 8, Nodes: 35
Depth 9, Nodes: 80
Depth 10, Nodes: 198
Depth 11, Nodes: 6
Depth 12, Nodes: 16
Depth 13, Nodes: 31
Depth 14, Nodes: 4
```

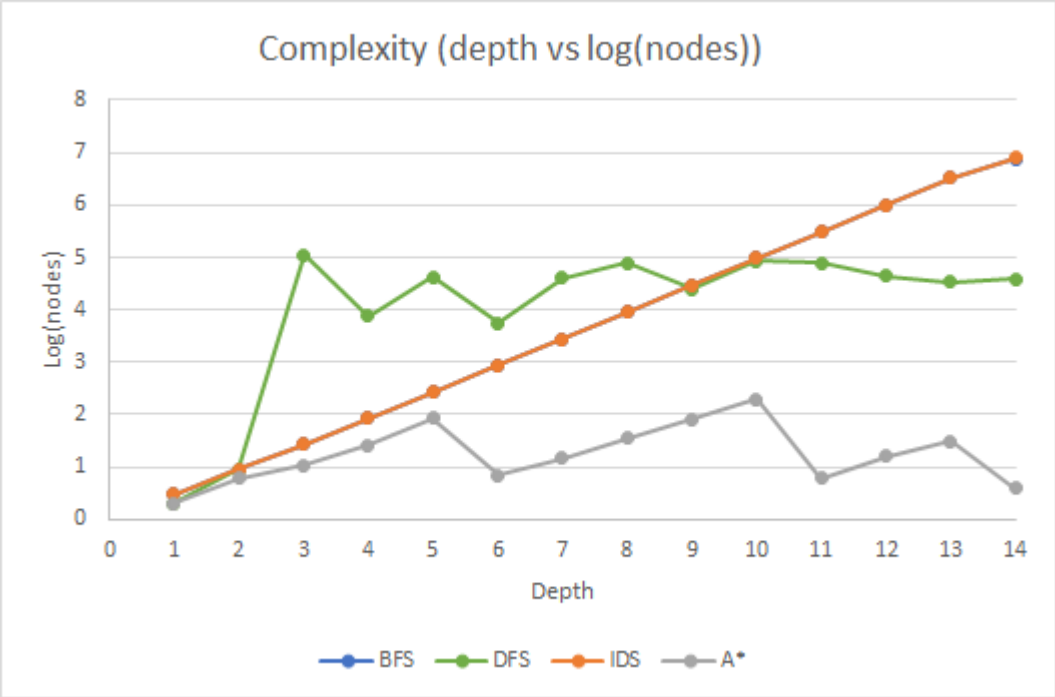
6.3 Complexity

6.3.1 Time complexity-graph



Depth	BFS nodes	DFS nodes	IDS nodes	A* nodes
1	3	2	3	2
2	9	9	9	6
3	27	109,304	27	11
4	85	7,652	85	26
5	271	42,421	271	84
6	873	5,590	873	7
7	2,819	39,816	2,819	15
8	9,117	78,912	9,117	35
9	29,495	24,936	29,495	80
10	95,441	87,274	95,441	198
11	308,843	78,412	308,843	6
12	999,429	45,032	999,429	16
13	3,234,207	33,698	3,234,207	31
14	7,694,972	39,250	8,000,849	4

6.3.2 Time complexity-log graph



Depth	BFS log	DFS log	IDS log	A* log
1	0.477121255	0.301029996	0.477121255	0.30103
2	0.954242509	0.954242509	0.954242509	0.77815125
3	1.431363764	5.038636055	1.431363764	1.04139269
4	1.929418926	3.883774961	1.929418926	1.41497335
5	2.432969291	4.627580902	2.432969291	1.92427929
6	2.941014244	3.747411808	2.941014244	0.84509804
7	3.450095076	4.600057628	3.450095076	1.17609126
8	3.959851955	4.897143051	3.959851955	1.54406804
9	4.469748401	4.396826789	4.469748401	1.90308999
10	4.979734981	4.940884881	4.979734981	2.29666519
11	5.489737762	4.894382531	5.489737762	0.77815125
12	5.999751947	4.653521236	5.999751947	1.20411998
13	6.509767813	4.527604126	6.509767813	1.49136169
14	6.886207044	4.593839661	6.903136074	0.60205999

6.4 Extension

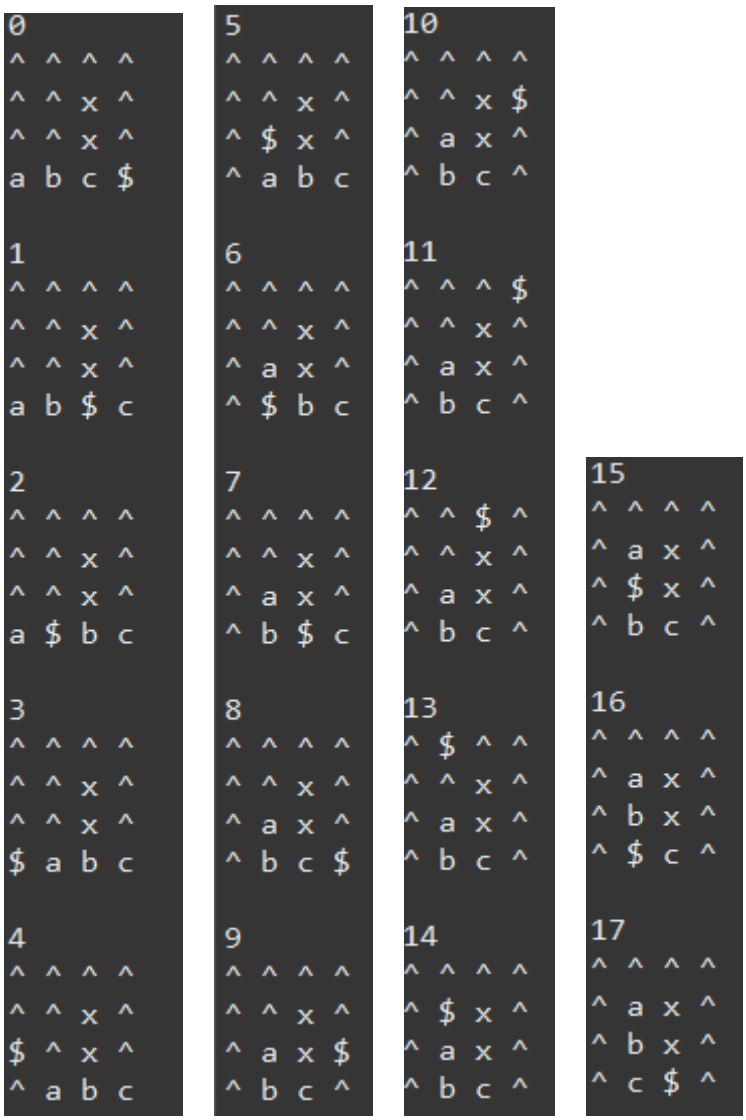
6.4.1 Obstacles - main

```
package blockwords;

import java.util.Collections;

public class Main {

    public static void main(String[] args) {
        char[][] startState = {
                {'^','^','^','^'},
                {'^','^','x','^'},
                {'^','^','x','^'},
                {'a','b','c','$'}
        };
        Node root = new Node(startState);

        Search search = new Search();
        List<Node> sol = search.AS(root);

        if(sol.size() > 0) {
            Collections.reverse(sol);
            for(int i = 0; i < sol.size(); i++) {
                System.out.println(i);
                sol.get(i).printGrid();
            }
        } else {
            System.out.println("No solution found");
        }
    }
}
```

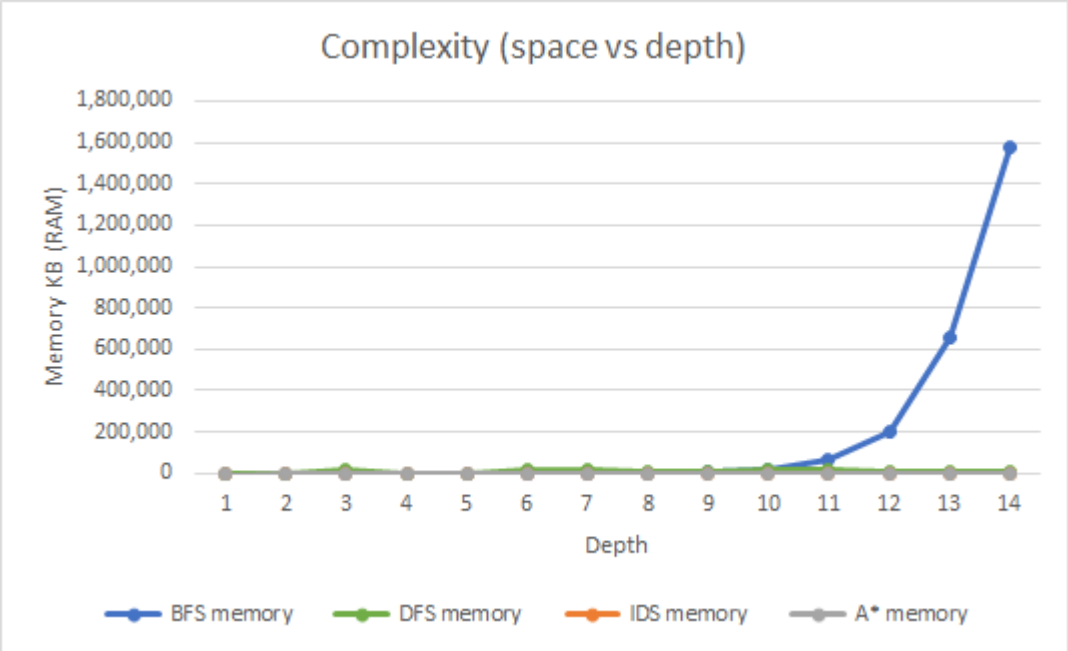
6.4.2 Obstacles A* - debug

```
Goal depth 17, Nodes expanded 1112
Depth 1, Nodes: 2
Depth 2, Nodes: 4
Depth 3, Nodes: 9
Depth 4, Nodes: 19
Depth 5, Nodes: 25
Depth 6, Nodes: 53
Depth 7, Nodes: 82
Depth 8, Nodes: 141
Depth 9, Nodes: 300
Depth 10, Nodes: 18
Depth 11, Nodes: 21
Depth 12, Nodes: 40
Depth 13, Nodes: 65
Depth 14, Nodes: 108
Depth 15, Nodes: 200
Depth 16, Nodes: 11
Depth 17, Nodes: 13
```

6.4.3 Obstacles - solution

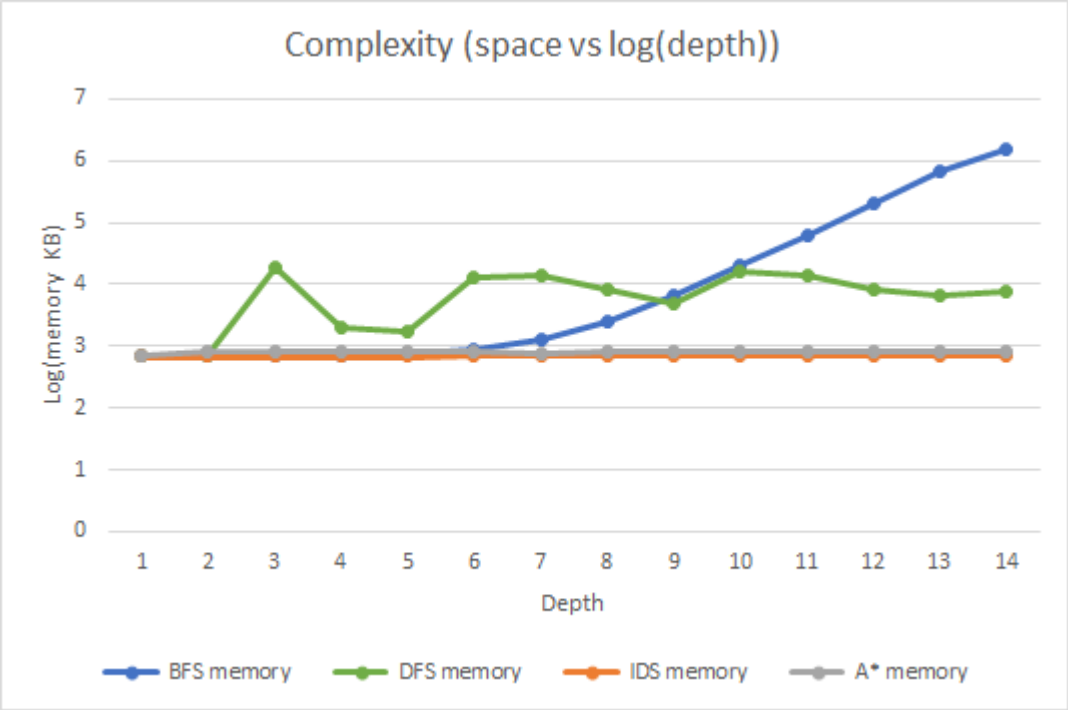


6.4.4 Space Complexity- graph



Depth	BFS memory	DFS memory	IDS memory	A* memory
1	714	718	716	713
2	715	721	716	795
3	719	19,504	716	796
4	730	2,033	716	796
5	768	1,679	716	817
6	891	12,847	717	813
7	1,289	14,281	717	748
8	2,576	8,071	718	817
9	6,738	5,004	718	819
10	20,209	15,718	718	826
11	63,800	14,195	718	821
12	204,862	8,458	718	824
13	661,348	6,510	718	834
14	1,572,526	7,464	721	857

6.4.5 Space Complexity- log graph



Depth	BFS memory	DFS memory	IDS memory	A* memory
1	2.853698212	2.856124444	2.854913022	2.85308953
2	2.854306042	2.857935265	2.854913022	2.90036713
3	2.85672889	4.290123688	2.854913022	2.90091307
4	2.86332286	3.308137379	2.854913022	2.90091307
5	2.88536122	3.225050696	2.854913022	2.91222206
6	2.949877704	4.108801724	2.855519156	2.91009055
7	3.110252917	4.154758619	2.855519156	2.8739016
8	3.410945859	3.906927347	2.856124444	2.91222206
9	3.828531007	3.699317301	2.856124444	2.9132839
10	4.305544824	4.196397284	2.856124444	2.91698005
11	4.804820679	4.152135397	2.856124444	2.91434316
12	5.311461408	3.927267681	2.856124444	2.91592721
13	5.820430045	3.813580989	2.856124444	2.92116605
14	6.196597835	3.872971631	2.857935265	2.93298082