INTELLIGENT SYSTEMS(COMP2208)

Search Methods

Radoslav Radomirov Enev(rre1u17)

ID: **29534771**

I.Approach

The whole project consists of 4 classes – **Main**, **State**, **Movement** and **Node**. **State** is a class that represents every single state of the grid. It also saves the information of the coordinates of the three building blocks, the agent and an array of blocked cells which can't be visited by the agent (this is an extension and it is optional). I have overridden the *toString* method so that later on I can visualize the whole state easier and also because it was easier for me to test if my searching methods were working properly.

The **Node** class represents the node of the tree. Every single node has its own state, a parent node, a level and depth. It has 2 key methods – *findPathTo* and *calculateDepthForAStar*. *findPathTo* finds the path to a given node(it is used to visualize the path of the search). *calculateDepthForAStar* uses the Manhattan Distance algorithm in order to calculate the distance between the current state of the node and the goal state. Moreover, in order to compare Nodes the Comparable interface is overridden.

The next class is **Movement**. It has a *move* method which moves the agent to a given node. There is also an *isBlocked* method which simply checks if a cell is blocked given x and y coordinates. There are 4 methods for moving left,right,up and down which use the *move* method and are very identical. They check if the agent can move to the given cell (if he tries to move out of the grid or the cell is blocked).

The final class is the **Main**. The Main class has a *main* method where the size of the grid, the start state and the goal state are initialized initialized. Furthermore, I wrote all 4 search methods in different methods in the **Main** class.

Depth-First Search takes a root node and a final state as arguments. It uses a stack data structure which it pops from every time and checks if the popped node is a goal state. If it doesn't find a solution, it expands the tree left, right, up and down in a randomized order every time and the children nodes are added to the stack. Same process is repeated until there is solution. DFS is not usually optimal but its main advantage is that it releases the memory it doesn't need.

Breadth-First Search is implemented using a queue. It removes the head of the queue and checks if it is the goal state. If it is not then it expands the current node and its children are added to the queue. This process repeats itself until there is a solution. The main advantage of BFS is that it always finds the most optimal solution, however, it is takes a lot of memory.)

Iterative Deepening Search simply runs DFS, but with limited, depth starting from 0, and it increases it every time if it doesn't find a solution. IDS always finds the most optimal solution. It's much faster than BFS and it also uses less memory.

A* Search uses a priority queue because the main idea of the strategy is to expand the nodes which have the lowest cost. To achieve that it uses the Manhattan Distance algorithm in order to calculate the lowest cost. First, it removes the head of the queue and it checks if a solution is found. If there is no solution found the current node is expanded and its children are added to the queue. This process is repeated until the solution is found. A* Search is always optimal, it expands the least number of nodes and is the fastest searching algorithm of all 4.

II.Evidence

II.1. DFS

Start state:

```
[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ]
[A] [B] [C] [*]
```

Popping and expanding:

```
[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ]
[A] [B] [*] [C]

[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ]
[A] [*] [B] [C]

[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ]
[ ] [*] [ ] [ ]
[A] [ ] [B] [C]

[ ] [ ] [ ] [ ]
[ ] [*] [ ] [ ]
[ ] [ ] [ ] [ ]
[A] [ ] [B] [C]

[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ]
[ ] [*] [ ] [ ]
[A] [ ] [B] [C]

[ ] [ ] [ ] [ ]
[ ] [*] [ ] [ ]
[ ] [ ] [ ] [ ]
[A] [ ] [B] [C]

```

```
[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ]
[ ] [*] [ ] [ ]
[A] [ ] [B] [C]
```

II.2. BFS

Start state:

```
[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ]
[A] [B] [C] [*]
```

Removing and expanding:

```
[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [*]
[A] [B] [C] [ ]

[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ]
[ ] [ ] [*] [ ]
[A] [B] [C] [ ]

[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ]
[ ] [*] [ ] [ ]
[A] [B] [C] [ ]

[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ]
[ ] [B] [ ] [ ]
[A] [*] [C] [ ]

[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ]
[ ] [B] [ ] [ ]
[*] [A] [C] [ ]
```

II.3. IDS

Start state: `[ ] [ ] [ ] [ ]`

```
[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ]
[A] [B] [C] [*]
```

Popping and expanding:

```
[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [*]
[A] [B] [C] [ ]

[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ]
[ ] [ ] [*] [ ]
[A] [B] [C] [ ]

[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ]
[ ] [*] [ ] [ ]
[A] [B] [C] [ ]

[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ]
[ ] [B] [ ] [ ]
[A] [*] [C] [ ]

[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ]
[ ] [B] [ ] [ ]
[*] [A] [C] [ ]

[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ]
[*] [B] [ ] [ ]
[ ] [A] [C] [ ]
```

II.4. A*

Start state:

```
[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ]
[A] [B] [C] [*]
```

Removing and expanding:

```
[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [*]
[A] [B] [C] [ ]

[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ]
[ ] [ ] [*] [ ]
[A] [B] [C] [ ]

[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ]
[ ] [*] [ ] [ ]
[A] [B] [C] [ ]

[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ]
[ ] [B] [ ] [ ]
[A] [*] [C] [ ]

[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ]
[ ] [B] [ ] [ ]
[*] [A] [C] [ ]
```

III. Scalability

Running all 4 searching algorithms on a 4x4 grid with the same position of all 3 building blocks and agent we can clearly see how different the results are.

Search name	Depth	Nodes expanded
BFS	14	6179534
DFS	23869	23869
IDS	14	3363438
A*	55	14773

The first things we notice are that the depth of DFS is equal to the nodes expanded. BFS on the other hand has expanded more than 6 million nodes but has a depth of 14. Furthermore, the results of DFS vary a lot. Sometimes I would get depth and nodes expanded ~ 4000 while other times I would get a number as big as 50000. We also notice that IDS has the same depth as BFS but almost twice less nodes expanded. Last

but not least, A* has only around 15000 nodes expanded. This is because it chooses to expand the node with the lowest cost.

Search name	Depth	Nodes expanded
BFS	out of memory	out of memory
DFS	59945	59945
IDS	14	19770858
A*	55	60515

We are looking at the results of the same algorithms with the exact same conditions except for the grid size which is 5x5 this time. BFS couldn't complete because it went out of memory.

Search name	Depth	Nodes expanded
BFS	out of memory	out of memory
DFS	3505722	3505722
IDS	14	42738253
A*	55	125564

This last table shows the results from a 6x6 grid. Clearly A* is the best algorithm our of the 4, because it's the fastest and it always finds an optimal solution. Then it is IDS which also always finds an optimal solution, but it is much slower than A* and if it's given problems with higher difficulty chances are that it will run out of space. BFS is the slowest out of the 4, however it also always finds an optimal solution. DFS can be better in certain situations but it doesn't always find an optimal solution.

IV. Extras and limitations

To show creativity I have added a way to add blocked tiles on the grid which the agent can't go to. This can create very interesting scenarios in which a solution is found faster, because some of the cells are blocked, but at the same time it can also create situations in which the agent gets stuck for some time, especially with breadth-first search where many nodes are getting expanded and each time the order of actions is randomized.

Moreover, I have made the grid resizable which can be used for further investigation of this task. The bigger the size of the grid the higher the difficulty of the problem becomes. There are more cells that needs to be explored.

The biggest drawback, in my opinion is that the code is not reusable. The search algorithms work only for this problem in particular. I also used arrays to store the coordinates of the blocks and the agent while a better approach would have been to create a separate **Cell** class where I keep all this data. In addition, changing the order of movements – left, right, up and down can have a huge impact on how fast a solution is found. In IDS, BFS and A* the movement is always done in the exact same order. I could have also added an option to add as many building blocks as a person wants. Right now there are always 3 building blocks and 1 agent and if you want to add another block you need to change the entire code which could have easily been avoided.

In conclusion, I can say that if I am doing this assignment now there are many other features I can add. I can also make the code a lot more reusable – less hard coded features and a lot more freedom, for example adding as many blocks as you can. Moreover, I am very confident that I would be able to solve the problem a lot faster with a much better implementation.

V. Code

V.1.Main.java

```java
import java.util.*;

/**
 * Main method for testing purposes plus all implementation of all 4 search methods
 */
public class Main {
    public static void main(String[] args) {
        // The size of the grid
        int size = 4;

        // Creating a new start state
        State startState = new State(size, new int[] {0,3}, new int[] {1,3}, new int[] {2, 3}, new int[] {3, 3});
        State goalState = new State(size, new int[] {1,1}, new int[] {1,2}, new int[] {1, 3}, null);

        // Adding the start state to a node
        Node node = new Node(startState);

        DepthFirstSearch(node, goalState);
        //BreadthFirstSearch(node, goalState);
        //IterativeDeepeningSearch(node, goalState);
        //AStarSearch(node, goalState);
    }

    /**
     * Compares the coordinates all blocks of the current state with the coordinates of the blocks of the goal state
     * @param state - current state to compare with goal
     * @param goal - the goal state
     * @return if the state is a goal state or not
     */
    private static boolean isGoal(State state, State goal) {
        boolean equalsBlockA = Arrays.equals(state.getBlockA(), goal.getBlockA());
        boolean equalsBlockB = Arrays.equals(state.getBlockB(), goal.getBlockB());
        boolean equalsBlockC = Arrays.equals(state.getBlockC(), goal.getBlockC());
        if (equalsBlockA && equalsBlockB && equalsBlockC){
            return true;
        }
        return false;
    }

    /**
     * Implementation of DFS
     * @param root - the root node of the state
     * @param goal - the goal state of the search
     */
```

```
45.     private static void DepthFirstSearch(Node root, State goal) {
46.         System.out.println("Depth-First Search on state:\n" + root.getState());
47.
48.         int nodes = 0;
49.         Stack<Node> stack = new Stack<>();
50.         Movement movement = new Movement();
51.
52.         stack.add(root);
53.
54.         while(!stack.isEmpty()) {
55.             ArrayList<Node> successors = new ArrayList<>();
56.             Node current = stack.pop();
57.
58.             // If the goal state is reached
59.             if(isGoal(current.getState(), goal)) {
60.                 ArrayList<Node> states = current.findPathTo(current);
61.
62.                 System.out.println("Depth-
    First Search has finished with depth: " + current.getDepth() + " | nodes expanded: " +
    nodes + "\n" + current.getState() + "\nStates:\n");
63.
64.                 for(Node state : states) {
65.                     System.out.println(state.getState());
66.                 }
67.                 break;
68.             }
69.
70.             successors.add(movement.goUp(current));
71.             successors.add(movement.goDown(current));
72.             successors.add(movement.goLeft(current));
73.             successors.add(movement.goRight(current));
74.             nodes++;
75.
76.             // Randomizing the moves
77.             Collections.shuffle(successors);
78.
79.             for(Node child : successors) {
80.                 if(child != null) {
81.                     stack.add(child);
82.                 }
83.             }
84.         }
85.     }
86.
87.     /**
88.      * Implementation of DFS
89.      * @param root - the root node of the state
90.      * @param goal - the goal state of the search
91.      */
92.     private static void BreadthFirstSearch(Node root, State goal) {
93.         System.out.println("Breadth-First Search on state:\n" + root.getState());
94.
95.         int nodes = 0;
96.         Queue<Node> queue = new LinkedList<>();
97.         Movement movement = new Movement();
98.
99.         queue.add(root);
100.
101.             while(!queue.isEmpty()) {
102.                 ArrayList<Node> successors = new ArrayList<>();
103.
```

```
104.                     Node current = queue.remove();
105.
106.                     // If the goal state is reached
107.                     if(isGoal(current.getState(), goal)) {
108.                         ArrayList<Node> steps = current.findPathTo(current);
109.
110.                         System.out.println("Breadth-
    First Search has finished with depth: " + current.getDepth() + " | nodes expanded: " +
    nodes + "\n" + current.getState() + "\nStates:\n");
111.
112.                         for(Node step : steps) {
113.                             System.out.println(step.getState());
114.                         }
115.                         break;
116.                     }
117.
118.                     successors.add(movement.goUp(current));
119.                     successors.add(movement.goDown(current));
120.                     successors.add(movement.goLeft(current));
121.                     successors.add(movement.goRight(current));
122.                     nodes++;
123.
124.                     for(Node child : successors) {
125.                         if(child != null) {
126.                             queue.add(child);
127.                         }
128.                     }
129.                 }
130.             }
131.
132.             /**
133.              * Implementation of IDS
134.              * @param root
135.              * @param goal
136.              */
137.             private static void IterativeDeepeningSearch(Node root, State goal) {
138.                 System.out.println("Breadth-
    First Search on state:\n" + root.getState());
139.
140.                 int nodes = 0;
141.                 int maxDepth = 0;
142.                 Stack<Node> stack = new Stack<>();
143.                 Movement movement = new Movement();
144.
145.                 stack.add(root);
146.
147.                 while(!stack.isEmpty()) {
148.                     ArrayList<Node> successors = new ArrayList<>();
149.
150.                     Node current = stack.pop();
151.
152.                     //If the goal state is reached
153.                     if(isGoal(current.getState(), goal)) {
154.                         ArrayList<Node> steps = current.findPathTo(current);
155.
156.                         System.out.println("Iterative deepening Search has finished with
     depth: " + current.getDepth() + " | nodes expanded: " + nodes + "\n" + current.getStat
    e() + "\nStates:\n");
157.
158.                         for(Node step : steps) {
159.                             System.out.println(step.getState());
```

```
                        }
                        break;
                    } else {
                        // Only expands the node if the nax depth of the node > current level
                        if(current.getLevel() < maxDepth) {
                            successors.add(movement.goUp(current));
                            successors.add(movement.goDown(current));
                            successors.add(movement.goLeft(current));
                            successors.add(movement.goRight(current));
                            nodes++;

                            for(Node child : successors) {
                                if(child != null) {
                                    stack.add(child);
                                }
                            }
                        }
                    }

                    // Increase the max depth if there is no solution for this depth
                    if(stack.size() == 0) {
                        stack.push(root);
                        maxDepth++;
                    }

                }
            }

            /**
             * Implementation of A*
             * @param root
             * @param goal
             */
            private static void AStarSearch(Node root, State goal) {
                System.out.println("A* Search on:\n" + root.getState());

                int nodes = 0;
                PriorityQueue<Node> queue = new PriorityQueue<>();
                Movement movement = new Movement();

                queue.add(root);
                while(!queue.isEmpty()) {

                    ArrayList<Node> successors = new ArrayList<>();
                    Node current = queue.poll();

                    //If the goal state is reached
                    if(isGoal(current.getState(), goal)) {

                        ArrayList<Node> steps = current.findPathTo(current);

                        System.out.println("A* deepening Search has finished with depth: " + current.getDepth() + " | nodes expanded: " + nodes + "\n" + current.getState() + "\nStates:\n");

                        for(Node step : steps) {
                            System.out.println(step.getState());
                        }
                        break;
                    }
```

```
218.
219.                     successors.add(movement.goUp(current));
220.                     successors.add(movement.goDown(current));
221.                     successors.add(movement.goLeft(current));
222.                     successors.add(movement.goRight(current));
223.                     nodes++;
224.
225.
226.                     for(Node child : successors) {
227.                         if(child != null) {
228.                             child.calculateDepthForAStar(goal);
229.                             queue.add(child);
230.                         }
231.                     }
232.                 }
233.             }
234.
235.         }
```

V.2.State.java

```java
import java.util.ArrayList;
/**
 * Represents a state of the grid
 */
public class State {

    /**
     * Size of the grid
     */
    private int gridSize;

    /**
     * The coordinates for the first building block
     */
    private int blockA[];

    /**
     * The coordinates for the second building block
     */
    private int blockB[];

    /**
     * The coordinates for the third building block
     */
    private int blockC[];

    /**
     * The coordinates for the agent
     */
    private int agent[];

    /**
     * A database with all blocked cells
     */
    private int blockedCells[][];

    /**
     * Creates a new state without any blocked cells
     * @param gridSize - the size of the grid
     * @param blockA - the coordinates of first building block
     * @param blockB - the coordinates of the second building block
     * @param blockC - the coordinates of the third building block
     * @param agent - the coordinates of the agent
     */
    public State(int gridSize, int blockA[], int blockB[], int blockC[], int agent[]) {

        this.gridSize = gridSize;
        this.blockA = blockA;
        this.blockB = blockB;
        this.blockC = blockC;
        this.agent = agent;
    }

```

```
53.     /**
54.      * Creates a state with blocked cells
55.      * @param gridSize - the size of the grid
56.      * @param blockA - the coordinates of first building block
57.      * @param blockB - the coordinates of the second building block
58.      * @param blockC - the coordinates of the third building block
59.      * @param agent - the coordinates of the agent
60.      * @param blockedCells - a database will all blocked cells
61.      */
62.     public State(int gridSize, int blockA[], int blockB[], int blockC[], int agent[], i
    nt[]...blockedCells) {
63.         this.gridSize = gridSize;
64.         this.blockA = blockA;
65.         this.blockB = blockB;
66.         this.blockC = blockC;
67.         this.agent = agent;
68.         this.blockedCells = blockedCells;
69.     }
70.
71.     /**
72.      * Gets the size of the grid
73.      * @return size of grid
74.      */
75.     public int getGridSize() {
76.         return gridSize;
77.     }
78.
79.     /**
80.      * Gets the coordinates of first block
81.      * @return first block cell
82.      */
83.     public int[] getBlockA() {
84.         return blockA;
85.     }
86.
87.     /**
88.      * Sets the coordinates of the first block
89.      * @param blockA - new cell
90.      */
91.     public void setBlockA(int blockA[]) {
92.         this.blockA = blockA;
93.     }
94.
95.     /**
96.      * Gets the coordinates of second block
97.      * @return second block cell
98.      */
99.     public int[] getBlockB() {
100.             return blockB;
101.         }
102.
103.         /**
104.          * Sets the coordinates of the second block
105.          * @param blockB
106.          */
107.         public void setBlockB(int blockB[]) {
108.             this.blockB = blockB;
109.         }
110.
111.         /**
112.          * Gets the coordinates of the third block
```

```
113.             * @return value of third block
114.             */
115.            public int[] getBlockC() {
116.                return blockC;
117.            }
118.
119.            /**
120.             * Gets the coordinates of the third block
121.             * @return the third block cell
122.             */
123.            public void setBlockC(int block[]) {
124.                this.blockC = blockC;
125.            }
126.
127.            /**
128.             * Gets the coordinates of the agent
129.             * @return the agent cell
130.             */
131.            public int[] getAgent() {
132.                return agent;
133.            }
134.
135.            /**
136.             * Sets a new value to the cell of the agent
137.             * @param agent - new cell of the agent
138.             */
139.            public void setAgent(int agent[]) {
140.                this.agent = agent;
141.            }
142.
143.            /**
144.             * Gets a list of all the blocked cells
145.             * @return
146.             */
147.            public int[][] getBlockedCells() {
148.                return blockedCells;
149.            }
150.
151.            /**
152.             * Modifies the toString method so that the current State is properly displa
    ced(This is only for our own convenience)
153.             * @return
154.             */
155.            @Override
156.            public String toString() {
157.                String val = "";
158.                for (int y = 0; y < gridSize; y++) {
159.                    for (int x = 0; x < gridSize; x++) {
160.                        if (x == getBlockA()[0] && y == getBlockA()[1]) {
161.                            val += "[A] ";
162.                        } else if (x == getBlockB()[0] && y == getBlockB()[1]) {
163.                            val += "[B] ";
164.                        } else if (x == getBlockC()[0] && y == getBlockC()[1]) {
165.                            val += "[C] ";
166.                        } else if (x == getAgent()[0] && y == getAgent()[1]) {
167.                            val += "[*] ";
168.                        } else {
169.                            boolean blocked = false;
170.                            if (blockedCells != null) {
171.                                for (int cell[] : blockedCells) {
172.                                    if (cell[0] == x && cell[1] == y) {
```

```
                                        val += "[!] ";
                                        blocked = true;
                                        break;
                                    }
                                }
                            }

                            if (!blocked) {
                                val += "[ ] ";
                            }
                        }
                    }
                    val += "\n";
                }
                return val;
            }

        }
```

V.3. Node.java

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

/**
 * Implements a tree node used later for all 4 searching algorithms
 */
public class Node implements Comparable<Node> {
    /**
     * The state of the node
     */
    private State state;

    /**
     * The parent node
     */
    private Node parent;

    /**
     * Level of the node
     */
    private int level;

    /**
     * The depth of this node
     */
    private int depth;

    /**
     * A node with a parent
     * @param state - the state of the node
     * @param parent - the parent of the node
     * @param level - the level of the node
     * @param depth - the cost of the node
     */
    public Node(State state, Node parent, int level, int depth) {
        setState(state);
        setParent(parent);
        setLevel(level);
        setDepth(depth);
    }

    /**
     * A node with no parent, root node
     * @param state - the state of the node
     */
    public Node(State state) {
        setState(state);
        setParent(null);
        setLevel(0);
        setDepth(0);
    }

    /**
     * Gets the current state
```

```
55.      * @return the state
56.      */
57.     public State getState() {
58.         return state;
59.     }
60.
61.     /**
62.      * Sets the state of the node
63.      * @param state - a new state
64.      */
65.     public void setState(State state) {
66.         this.state = state;
67.     }
68.
69.     /**
70.      * Gets the parent of the node
71.      * @return parent
72.      */
73.     public Node getParent() {
74.         return parent;
75.     }
76.
77.     /**
78.      * Sets the parent of the node
79.      * @param parent - the parent node
80.      */
81.     public void setParent(Node parent) {
82.         this.parent = parent;
83.     }
84.
85.     /**
86.      * Gets the level of the node
87.      * @return
88.      */
89.     public int getLevel() {
90.         return level;
91.     }
92.
93.     /**
94.      * Sets the level of the node
95.      * @param level - new level
96.      */
97.     public void setLevel(int level) {
98.         this.level = level;
99.     }
100.
101.          /**
102.           * Gets the cost of the node
103.           * @return
104.           */
105.          public int getDepth() {
106.              return depth;
107.          }
108.
109.          /**
110.           * Sets the cost of the node
111.           * @param cost
112.           */
113.          public void setDepth(int cost) {
114.              this.depth = cost;
115.          }
```

```
116.
117.            /**
118.             * Finds to path to a given node
119.             * @param toFind - the node to find the path to
120.             * @return the path to the node
121.             */
122.            public ArrayList<Node> findPathTo(Node toFind) {
123.                ArrayList<Node> path = new ArrayList<>();
124.                while (toFind.parent != null) {
125.                    path.add(toFind);
126.                    toFind = toFind.parent;
127.                }
128.                Collections.reverse(path);
129.                return path;
130.            }
131.
132.            /**
133.             * Calculates the depth between the current state and the goal state(needed
    for A*)
134.             * @param goal - goal state
135.             */
136.            public void calculateDepthForAStar(State goal) {
137.                int distanceFromGoalA = Math.abs(state.getBlockA()[0] - goal.getBlockA()
    [0]) + Math.abs(state.getBlockA()[1] - goal.getBlockA()[1]);
138.                int distanceFromGoalB = Math.abs(state.getBlockB()[0] - goal.getBlockB()
    [0]) + Math.abs(state.getBlockB()[1] - goal.getBlockB()[1]);
139.                int distanceFromGoalC = Math.abs(state.getBlockC()[0] - goal.getBlockC()
    [0]) + Math.abs(state.getBlockC()[1] - goal.getBlockC()[1]);
140.                depth += distanceFromGoalA + distanceFromGoalB + distanceFromGoalC;
141.            }
142.
143.            @Override
144.            public int compareTo(Node node) {
145.                if (getDepth() - node.getDepth() == 0) {
146.                    if (getLevel() > node.getLevel()) {
147.                        return -1;
148.                    } else if(node.getLevel() == getLevel()) {
149.                        return 0;
150.                    } else {
151.                        return 1;
152.                    }
153.                }
154.                return getDepth() - node.getDepth();
155.            }
156.        }
```

V.4. Movement.java

```
1.  import java.util.Arrays;
2.
3.  public class Movement {
4.
5.      /**
6.       * The main logic behind the movement is done here.
7.       * @param node - the node which the movements needs to be done on
8.       * @param newAgent - the new position of the agent
9.       * @return the node that it was moved to
10.      */
11.     private Node move(Node node, int newAgent[]) {
12.         Node moveTo;
13.         if(Arrays.equals(node.getState().getBlockA(), newAgent)) {
14.             State state = new State(node.getState().getGridSize(), node.getState().getA
    gent(), node.getState().getBlockB(), node.getState().getBlockC(), newAgent, node.getSta
    te().getBlockedCells());
15.             moveTo = new Node(state, node, node.getLevel() + 1, node.getDepth() + 1);
16.         } else if(Arrays.equals(node.getState().getBlockB(), newAgent)) {
17.             State state = new State(node.getState().getGridSize(), node.getState().getB
    lockA(), node.getState().getAgent(), node.getState().getBlockC(), newAgent, node.getSta
    te().getBlockedCells());
18.             moveTo = new Node(state, node, node.getLevel() + 1, node.getDepth() + 1);
19.         } else if(Arrays.equals(node.getState().getBlockC(), newAgent)) {
20.             State state = new State(node.getState().getGridSize(), node.getState().getB
    lockA(), node.getState().getBlockB(), node.getState().getAgent(), newAgent, node.getSta
    te().getBlockedCells());
21.             moveTo = new Node(state, node, node.getLevel() + 1, node.getDepth() + 1);
22.         } else {
23.             State state = new State(node.getState().getGridSize(), node.getState().getB
    lockA(), node.getState().getBlockB(), node.getState().getBlockC(), newAgent, node.getSt
    ate().getBlockedCells());
24.             moveTo = new Node(state, node, node.getLevel() + 1, node.getDepth() + 1);
25.         }
26.         return moveTo;
27.     }
28.
29.     /**
30.      * Checks if a cell exists in the blocked cells database
31.      * @param x - x of the blocked cell
32.      * @param y - y of the blocked cell
33.      * @param blockedCells - database with blocked cells and their coordinates
34.      * @return if the cell on the given coordinates is blocked or not
35.      */
36.     private boolean isBlocked(int x, int y, int blockedCells[][]) {
37.         boolean blocked = false;
38.         if (blockedCells != null) {
39.             for (int[] cell : blockedCells) {
40.                 if (x == cell[0] && y == cell[1]) {
41.                     blocked = true;
42.                     break;
43.                 }
44.             }
45.         }
46.         return blocked;
```

```
47.     }
48.
49.     /**
50.      * Moves the node up
51.      * @param node - the node which is being moved
52.      * @return the new node after the movement is done, if the node is blocked - don't
    do anything
53.      */
54.     public Node goUp(Node node) {
55.         if (node.getState().getAgent()[1] - 1 >= 0 && !isBlocked(node.getState().getAge
    nt()[0], node.getState().getAgent()[1] - 1, node.getState().getBlockedCells()))  {
56.             return move(node, new int[] {node.getState().getAgent()[0], node.getState()
    .getAgent()[1] - 1});
57.         }
58.         return null;
59.     }
60.
61.     /**
62.      * Moves the node down
63.      * @param node - the node which is being moved
64.      * @return the new node after the movement is done, if the node is blocked - don't
    do anything
65.      */
66.     public Node goDown(Node node) {
67.         if (node.getState().getAgent()[1] + 1 <= node.getState().getGridSize() - 1&& !i
    sBlocked(node.getState().getAgent()[0], node.getState().getAgent()[1] + 1, node.getStat
    e().getBlockedCells()) ) {
68.             return move(node, new int[] {node.getState().getAgent()[0], node.getState()
    .getAgent()[1] + 1});
69.         }
70.         return null;
71.     }
72.
73.     /**
74.      * Moves the node to the left
75.      * @param node - the node which is being moved
76.      * @return the new node after the movement is done, if the node is blocked - don't
    do anything
77.      */
78.     public Node goLeft(Node node) {
79.         if (node.getState().getAgent()[0] - 1 >= 0 && !isBlocked(node.getState().getAge
    nt()[0] - 1, node.getState().getAgent()[1], node.getState().getBlockedCells()) ) {
80.             return move(node, new int[] {node.getState().getAgent()[0] - 1, node.getSta
    te().getAgent()[1]});
81.         }
82.         return null;
83.     }
84.
85.     /**
86.      * Moves the node to the right
87.      * @param node - the node which is being moved
88.      * @return the new node after the movement is done, if the node is blocked - don't
    do anything
89.      */
90.     public Node goRight(Node node) {
91.         if (node.getState().getAgent()[0] + 1 <= node.getState().getGridSize() - 1 && !
    isBlocked(node.getState().getAgent()[0] + 1, node.getState().getAgent()[1], node.getSta
    te().getBlockedCells()) ) {
92.             return move(node, new int[] {node.getState().getAgent()[0] + 1, node.getSta
    te().getAgent()[1]});
93.         }
```

```
94.         return null;
95.     }
96.
97. }
```