COMP2208 INTELLIGENT SYSTEMS

SEARCH METHODS

December 1, 2016

Vedant Chokshi
Student ID: 27748456
University of Southampton

Contents

1 Approach **3**

2 Evidence **4**

2.1 Breadth-First Search Evidence 5

2.2 Depth-First Search Evidence 6

2.3 Iterative Deepening Search Evidence 7

2.4 A* Search Evidence 8

3 Scalability **9**

4 Extras and Limitations **11**

5 Code **12**

5.1 State.java 12

5.2 Node.java 15

5.3 Move.java 18

5.4 Search.java 22

1 APPROACH

Firstly, I approached this problem by creating a **State** class which represents the current state of my Blocksworld Puzzle. The **State** class consists of the size of the grid, the position of Tile A, the position of Tile B, the position of Tile C, the position of the Agent (Tile #) and a list of blocked tiles which cannot be visited (which is optional). I then created a class **Node** which represents a node of the tree that consists of the **State**, the level of the node in the tree, its parent **Node** and the cost to get to the Node. Next, I created a class **Move** which consisted of methods to simulate *up, down, left, right* on a given **Node** and return a new **Node** based on the movement. After implementing the basic structure of the Blocksworld Puzzle, I created a **Search** class which consisted of all the search methods and runs them. The while loop continues until a solution is found.

The Breadth-First Search (BFS) method works by taking in the root **Node** of the tree (starting state) and a goal **State**. As BFS uses a *first in, first out (FIFO)* structure, I decided use the **Queue** data structure in java as my fringe. This works by removing the head of the **Queue** and checking if the the removed **Node** is a solution. If not, the **Node** is expanded and its children are added to the queue. This process is constantly repeated until a solution is found. BFS finds the most optimal solution but takes up the greatest amount of memory compared to other searches.

The Depth-First Search (DFS) method works by taking in the root **Node** of the tree (starting state) and a goal **State**. As DFS uses a *last in, first out (LIFO)* structure, I decided use the **Stack** data structure in java as my fringe. DFS works by popping the top of the **Stack** and checking if the removed **Node** is a solution. If not, the Node is expanded (directions picked randomly) and its children are added to the **Stack**. This process is repeated until a solution is found. DFS very rarely finds the most optimal solution but tends to use a low amount of memory.

The Iterative Deepening Search (IDS) method works by taking in the root **Node** of the tree (starting state) and a goal **State**. It works by running DFS with a limited depth while iteratively increasing the depth and checking if there is a solution on each depth. The depth starts at 0 and is incremented for each iterations until the solution is found. IDS finds the most optimal solution and is much better than BFS as it is quicker and takes up less memory.

The A* search method works by taking in the root **Node** of the tree (starting state) and a goal **State**. As the idea of A* is to expand nodes that have the lowest cost, I decided use the **Priority Queue** data structure in java as my fringe which puts Nodes with the lowest cost (calculated using current cost + Manhattan Distance) at the front of the queue. A* by removing the head of the **Priority Queue** and checking if the **Node** removed is a solution. If not, the **Node** is expanded and its children are added to the **Priority Queue**. This process is repeated until a solution is found. A* finds the most optimal solution in the least amount of time with the least number of Nodes expanded.

2 EVIDENCE

For proving purposes of the search methods, I have set an easy problem difficulty with the optimal solution depth of the problem to 1 and altered my code output so the searches can be visualised.

Section 2.1 shows the altered output of the Breadth-First Search. It can be seen that the the root **Node** is expanded in to 4 directions which creates 4 new nodes. These nodes are added to the queue in the order they are made (*up, down, left, right*). It can be seen that the left move (3rd **Node** expanded from root) on the root is the optimal solution but BFS first removes and expands the up and down move for the root as they are added to the queue first. When the queue removes the left move from the queue, it can be seen that it is the solution so the search is finished.

Section 2.2 shows the altered output of the Depth-First Search. It can be seen that the the root **Node** is expanded in to 4 directions which creates 4 new nodes. These nodes are added to the stack in a random order (but in this case *up,left,down,right).* It can be seen that the left move (2nd **Node** expanded from root) on the root is the optimal solution but the top of the stack contains the right move (4th **Node** expanded from root). It then repeats this cycle (pop from the stack, expand randomly, add to stack) until the the top that is popped off is the solution.

Section 2.3 shows the altered output of the Iterative-Deepening Search. It can be seen that the the root **Node** is not the solution and cannot expand as the root level = max depth. The stack is also empty so the search is started again the the max depth incremented by one. In this case, the root is expanded. Each expanded **Node** is then popped but cannot be expanded further as popped **Node** depth = max depth. Eventually the second node to be popped off the stack is the solution.

Section 2.4 shows the altered output of the A* Search. It can be seen that the root is expanded in to 4 directions which creates 4 new nodes. The cost is calculated for the 4 new nodes and they are added to the priority queue where the **Nodes** are arranged by their cost. The first **Node** to be removed from the queue is the **Node** with the lowest cost (calculated using current cost + Manhattan Distance). The removed **Node** with the cost = 1 is the solution.

2.1 Breadth-First Search Evidence

```
Root added:
[ ] [ ] [ ] [ ]
[ ] [A] [ ] [ ]
[B] [#] [ ] [ ]
[ ] [C] [ ] [ ]

Removing:           Expanding:
[ ] [ ] [ ] [ ]     [ ] [ ] [ ] [ ]     [ ] [ ] [ ] [ ]
[ ] [A] [ ] [ ]     [ ] [#] [ ] [ ]     [ ] [A] [ ] [ ]
[B] [#] [ ] [ ]     [B] [A] [ ] [ ]     [B] [C] [ ] [ ]
[ ] [C] [ ] [ ]     [ ] [C] [ ] [ ]     [ ] [#] [ ] [ ]

                    [ ] [ ] [ ] [ ]     [ ] [ ] [ ] [ ]
                    [ ] [A] [ ] [ ]     [ ] [A] [ ] [ ]
                    [#] [B] [ ] [ ]     [B] [ ] [#] [ ]
                    [ ] [C] [ ] [ ]     [ ] [C] [ ] [ ]

Removing:           Expanding:
[ ] [ ] [ ] [ ]     [ ] [#] [ ] [ ]     [ ] [ ] [ ] [ ]
[ ] [#] [ ] [ ]     [ ] [ ] [ ] [ ]     [ ] [A] [ ] [ ]
[B] [A] [ ] [ ]     [B] [A] [ ] [ ]     [B] [#] [ ] [ ]
[ ] [C] [ ] [ ]     [ ] [C] [ ] [ ]     [ ] [C] [ ] [ ]

                    [ ] [ ] [ ] [ ]     [ ] [ ] [ ] [ ]
                    [#] [ ] [ ] [ ]     [ ] [ ] [#] [ ]
                    [B] [A] [ ] [ ]     [B] [A] [ ] [ ]
                    [ ] [C] [ ] [ ]     [ ] [C] [ ] [ ]

Removing:           Expanding:
[ ] [ ] [ ] [ ]     [ ] [ ] [ ] [ ]     [ ] [ ] [ ] [ ]
[ ] [A] [ ] [ ]     [ ] [A] [ ] [ ]     [ ] [A] [ ] [ ]
[B] [C] [ ] [ ]     [B] [#] [ ] [ ]     [B] [C] [ ] [ ]
[ ] [#] [ ] [ ]     [ ] [C] [ ] [ ]     [#] [ ] [ ] [ ]

                    [ ] [ ] [ ] [ ]
                    [ ] [A] [ ] [ ]
                    [B] [C] [ ] [ ]
                    [ ] [ ] [#] [ ]


Removing:
[ ] [ ] [ ] [ ]
[ ] [A] [ ] [ ]
[#] [B] [ ] [ ]
[ ] [C] [ ] [ ]
Solution found!
```

2.2 Depth-First Search Evidence

```
Root added:
[ ] [ ] [ ] [ ]
[ ] [A] [ ] [ ]
[B] [#] [ ] [ ]
[ ] [C] [ ] [ ]

Popping:            Expanding:
[ ] [ ] [ ] [ ]     [ ] [ ] [ ] [ ]     [ ] [ ] [ ] [ ]
[ ] [A] [ ] [ ]     [ ] [#] [ ] [ ]     [ ] [A] [ ] [ ]
[B] [#] [ ] [ ]     [B] [A] [ ] [ ]     [#] [B] [ ] [ ]
[ ] [C] [ ] [ ]     [ ] [C] [ ] [ ]     [ ] [C] [ ] [ ]

                    [ ] [ ] [ ] [ ]     [ ] [ ] [ ] [ ]
                    [ ] [A] [ ] [ ]     [ ] [A] [ ] [ ]
                    [B] [C] [ ] [ ]     [B] [ ] [#] [ ]
                    [ ] [#] [ ] [ ]     [ ] [C] [ ] [ ]

Popping:            Expanding:
[ ] [ ] [ ] [ ]     [ ] [ ] [ ] [ ]     [ ] [ ] [ ] [ ]
[ ] [A] [ ] [ ]     [ ] [A] [#] [ ]     [ ] [A] [ ] [ ]
[B] [ ] [#] [ ]     [B] [ ] [ ] [ ]     [B] [ ] [ ] [ ]
[ ] [C] [ ] [ ]     [ ] [C] [ ] [ ]     [ ] [C] [#] [ ]

                    [ ] [ ] [ ] [ ]     [ ] [ ] [ ] [ ]
                    [ ] [A] [ ] [ ]     [ ] [A] [ ] [ ]
                    [B] [ ] [ ] [#]     [B] [#] [ ] [ ]
                    [ ] [C] [ ] [ ]     [ ] [C] [ ] [ ]

Popping:            Expanding:
[ ] [ ] [ ] [ ]     [ ] [ ] [ ] [ ]     [ ] [ ] [ ] [ ]
[ ] [A] [ ] [ ]     [ ] [A] [ ] [ ]     [ ] [A] [ ] [ ]
[B] [#] [ ] [ ]     [B] [C] [ ] [ ]     [B] [ ] [#] [ ]
[ ] [C] [ ] [ ]     [ ] [#] [ ] [ ]     [ ] [C] [ ] [ ]

                    [ ] [ ] [ ] [ ]     [ ] [ ] [ ] [ ]
                    [ ] [#] [ ] [ ]     [ ] [A] [ ] [ ]
                    [B] [A] [ ] [ ]     [#] [B] [ ] [ ]
                    [ ] [C] [ ] [ ]     [ ] [C] [ ] [ ]


Popping:
[ ] [ ] [ ] [ ]
[ ] [A] [ ] [ ]
[#] [B] [ ] [ ]
[ ] [C] [ ] [ ]
Solution found!
```

2.3 Iterative Deepening Search Evidence

```
Root added:
[ ] [ ] [ ] [ ]
[ ] [A] [ ] [ ]
[B] [#] [ ] [ ]
[ ] [C] [ ] [ ]

Popping:
[ ] [ ] [ ] [ ]
[ ] [A] [ ] [ ]
[B] [#] [ ] [ ]
[ ] [C] [ ] [ ]

Cannot expand node as node level = maxDepth (0)
Stack empty! No solution found on depth 0. Increasing depth to 1...

Root added:
[ ] [ ] [ ] [ ]
[ ] [A] [ ] [ ]
[B] [#] [ ] [ ]
[ ] [C] [ ] [ ]

Popping:            Expanding:
[ ] [ ] [ ] [ ]     [ ] [ ] [ ] [ ]     [ ] [ ] [ ] [ ]
[ ] [A] [ ] [ ]     [ ] [#] [ ] [ ]     [ ] [A] [ ] [ ]
[B] [#] [ ] [ ]     [B] [A] [ ] [ ]     [B] [C] [ ] [ ]
[ ] [C] [ ] [ ]     [ ] [C] [ ] [ ]     [ ] [#] [ ] [ ]

                    [ ] [ ] [ ] [ ]     [ ] [ ] [ ] [ ]
                    [ ] [A] [ ] [ ]     [ ] [A] [ ] [ ]
                    [#] [B] [ ] [ ]     [B] [ ] [#] [ ]
                    [ ] [C] [ ] [ ]     [ ] [C] [ ] [ ]

Popping:
[ ] [ ] [ ] [ ]
[ ] [A] [ ] [ ]
[B] [ ] [#] [ ]
[ ] [C] [ ] [ ]

Cannot expand node as node level = maxDepth (1)

Popping:
[ ] [ ] [ ] [ ]
[ ] [A] [ ] [ ]
[#] [B] [ ] [ ]
[ ] [C] [ ] [ ]
Solution found!
```

2.4 A* Search Evidence

```
Root added:
[ ] [ ] [ ] [ ]
[ ] [A] [ ] [ ]
[B] [#] [ ] [ ]
[ ] [C] [ ] [ ]

Removing:
[ ] [ ] [ ] [ ]
[ ] [A] [ ] [ ]
[B] [#] [ ] [ ]
[ ] [C] [ ] [ ]

Expanding:
[ ] [ ] [ ] [ ]
[ ] [#] [ ] [ ]
[B] [A] [ ] [ ]
[ ] [C] [ ] [ ]
cost =   3

[ ] [ ] [ ] [ ]
[ ] [A] [ ] [ ]
[#] [B] [ ] [ ]
[ ] [C] [ ] [ ]
cost =   1

[ ] [ ] [ ] [ ]
[ ] [A] [ ] [ ]
[B] [C] [ ] [ ]
[ ] [#] [ ] [ ]
cost = 3

[ ] [ ] [ ] [ ]
[ ] [A] [ ] [ ]
[B] [ ] [#] [ ]
[ ] [C] [ ] [ ]
cost = 2

Removing:
[ ] [ ] [ ] [ ]
[ ] [A] [ ] [ ]
[#] [B] [ ] [ ]
[ ] [C] [ ] [ ]
Solution found!
```

3 SCALABILITY

To control the problem difficulty, I ran the the search methods on different problem complexities. I did this by moving the start state towards the final state by moving along the optimal path which was calculated using the other searches. I recorded the number of nodes expanded to reach solution by each search for the optimal solution depth of problem from 0-14. As the results of DFS vary, I ran the DFS 5000 times on each of the problem and calculated the average. Below are my results separated into 2 separate graphs to give a more accurate visualisation.

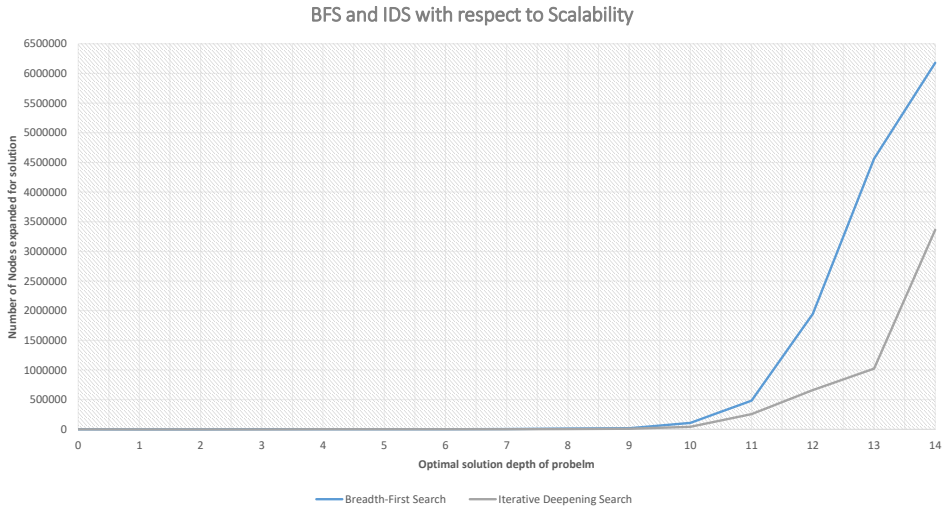

Figure 1: Graph showing the number of nodes expanded to reach the solution with respect to the optimal solution depth for BFS and IDS

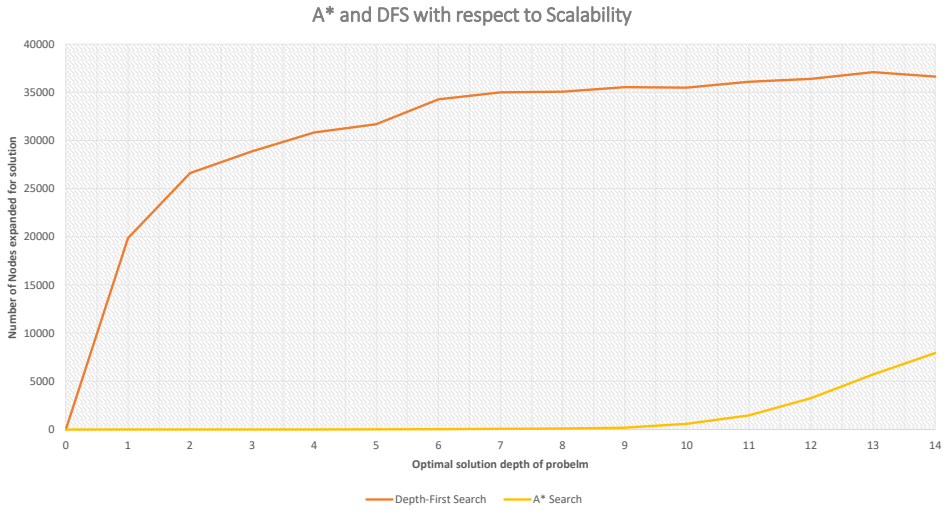

Figure 2: Graph showing the number of nodes expanded to reach the solution with respect to the optimal solution depth for DFS and A*

From Figure 1, it can be seen that the computational time of IDS is much lower than the computational time BFS. Even though IDS searches through a lot more nodes than BFS, it doesn't expand all of them. This is largely due to that fact that IDS doesn't expand all the nodes on the maximum depth unlike BFS.

From Figure 2, it can be seen that the computational time of A* is much lower than the computational time for DFS. This is largely due to the fact that DFS does not find the optimal solution and constantly expands the deepest unexpanded node until a solution is found. On the other hand, A* uses the a stic function to expand the node with the lowest cost (node which is closest to goal) and therefore expands the least amount of nodes to reach the solution.

When comparing all 4 searches in respect to the computational time, I can conclude that A* is the most optimum search method as it expands the least amount of nodes. A* is then followed by DFS, IDS and lastly BFS. Even though DFS expands the second best computational time, it very rarely returns an optimal solution which in a way does not make it a very good search method to use to find an optimal solution. IDS also finds the optimum solution but takes longer than A* as IDS searches and expands nodes to the max depth for each iteration until the solution is found. I feel like for much bigger problem sets, IDS would be able to find the solution as it does not run out of heap space unlike A*. Similarly to A* and IDS, BFS also finds the optimum solution however it expands the most amount of nodes which means it is the slowest. Like A*, BFS will also not work with bigger problem sets as it will run out of heap space.

4 EXTRAS AND LIMITATIONS

To show evidence of creativity/additional technical challenge/ambition, I decided to implement 3 extensions into my coursework. My first extension is having the ability to have a re-sizable grid. This means that the user has the ability to pick the size of the grid on which the tree search is run on. Increasing the grid size increased the problem difficulty (as the search space is greater) for the BFS, DFS, IDS and A*. However a limitation to this is the fact that the BFS and A* methods will not the run for higher problems as they will begin to run out of heap space as all nodes are stored in the memory.

Another extension I implemented is the ability to add multiple tiles. I did this by using Java varargs which allowed user to enter multiple coordinates for tiles while need to be blocked. The biggest limitation to this extension was the repercussion of blocking tile(s). Blocking certain tile(s) could result in a quicker solution by reducing the problem space or could result in a longer solution or even no solution in some cases. This makes it very hard to test as the problem difficulty is very difficult to control.

For my final extension, I implemented is to tie-break within my A* search to further optimise it. When printing out the cost for each node when testing A*, I realised that many nodes have the same stic and the best node isn't always picked. I resolved this my implementing a check in priority queue comparator which compares the levels of the nodes with the same cost and puts the the nodes with the highest level at the front of the queue. This resulted in my number of nodes expanded for A* to change from 17978 to 7941.

One limitation of my code is that the number of nodes expanded for A*, IDS and BFS are dependant on the order the nodes are expanded in. If the order is changed then the computational time could be a lot different to the current movement of *up, down, left, right*. Even for the optimised A* tie breaker, there are times when there are nodes which are on the same level and have the same stic. In these cases the order they are put into the fringe affect which node is expanded. In short, the limitation is that there is no order for which the nodes can be expanded so that the minimum number of nodes are expanded for A*, IDS and BFS is as small for possible.

Another limitation of my code is that I believe it is not reusable, the tree search methods only work for the Node defined to work for the 'Blocksworld Puzzle'. I have also hardcoded many parts of the puzzle like only allowing the user to have Tiles A, B, C and (Agent) so no more Tiles can be added to increase difficulty of the problem.

To summarise, I believe there are some drawbacks to my code and if I was to re-create this puzzle again, I would try to ensure that my code is not limited to the limitations I stated above.

5 CODE

5.1 State.java

```
/**
 * State Class implements the basic structure of a State of the
    Puzzle
 * @author Vedant Chokshi
 *
 */
public class State {
   private int[] gridSize, positionA, positionB, positionC,
      positionAgent;
   private int[][] blockedTiles;

   /**
    * Create a State without blocked tiles
    * @param gridSize Grid Size
    * @param positionA Position of A
    * @param positionB Position of B
    * @param positionC Position of C
    * @param positionAgent Position Agent
    */
   public State(int[] gridSize, int[] positionA, int[] positionB,
      int[] positionC, int[] positionAgent) {
      this.gridSize = gridSize;
      this.positionA = positionA;
      this.positionB = positionB;
      this.positionC = positionC;
      this.positionAgent = positionAgent;
   }

   /**
    * Create a State without blocked tiles
    * @param gridSize Grid Size
    * @param positionA Position of A
    * @param positionB Position of B
    * @param positionC Position of C
    * @param positionAgent Position Agent
```

```
     * @param blockedTiles Array of blocked tiles
     */
    public State(int[] gridSize, int[] positionA, int[] positionB,
        int[] positionC, int[] positionAgent, int[]...blockedTiles)
        {
        this.gridSize = gridSize;
        this.positionA = positionA;
        this.positionB = positionB;
        this.positionC = positionC;
        this.positionAgent = positionAgent;
        this.blockedTiles = blockedTiles;
    }

    /**
     * @return Grid Size
     */
    public int[] getGridSize() {
        return gridSize;
    }

    /**
     * @return Return position A
     */
    public int[] getPositionA() {
        return positionA;
    }

    /**
     * @return Return position B
     */
    public int[] getPositionB() {
        return positionB;
    }

    /**
     * @return Return position C
     */
    public int[] getPositionC() {
```

```
    return positionC;
}

/**
 * @return Return position Agent
 */
public int[] getPositionAgent() {
    return positionAgent;
}

/**
 * @return Return all the blocked tiles
 */
public int[][] getBlockedTiles() {
    return blockedTiles;
}

/**
 * Print out the node in a set format
 */
public String toString() {
    String output = "";

      for (int y = 1; y <= gridSize[0]; y++) {
          for (int x = 1; x <= gridSize[1]; x++) {
              if(x == positionA[0] && y == positionA[1]) {
                output += "[A] ";
              } else if(x == positionB[0] && y == positionB[1])
                 {
                output += "[B] ";
              } else if(x == positionC[0] && y == positionC[1])
                 {
                output += "[C] ";
              } else if(x == positionAgent[0] && y ==
                 positionAgent[1]) {
                output += "[#] ";
              } else {
                boolean flag = false;
```

```
                    if(blockedTiles != null) {
                        for(int[] coord : blockedTiles) {
                            if(coord[0] == x && coord[1] == y) {
                                output += "[-] ";
                                flag = true;
                                break;
                            }
                        }
                    }
                    if(!flag) {
                        output += "[ ] ";
                    }
                }
            }
            output += "\n";
        }
        return output;
    }
}
```

5.2 Node.java

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

/**
 * Node class implements the basic structure of a node in the
    tree which contains the configuration (state), the parent,
    the level the node is on and the cost to get to the Node
 * @author Vedant Chokshi
 */
public class Node implements Comparable<Node> {
    private int level, cost;
    private State state;
    private Node parent = null;

    /**
```

```
     * When instantiating a Node, if just give a state, then
        assume that it is the root node of a tree and set parent
        to null , node level to 0 and the cost to 0
     * @param state The state in the puzzle
     */
    public Node(State state) {
        this.state = state;
        this.parent = null;
        this.level = 0;
        this.cost = 0;
    }

    /**
     * Creates a Node with a state, parent and level
     * @param state The state in the puzzle
     * @param parent Parent of the Node
     * @param level Level of the Node
     * @param cost Cost to get the Node
     */
    public Node(State state, Node parent, int level, int cost) {
        this.state = state;
        this.parent = parent;
        this.level = level;
        this.cost = level;
    }

    /**
     * @return Current level of the Node
     */
    public int getLevel() {
        return level;
    }

    /**
     * @return Cost of the Node
     */
    public int getCost() {
        return cost;
```

```
    }

    /**
     * @return State of the puzzle in the Node
     */
    public State getState() {
        return state;
    }

    /**
     * Calculates the cost using the Manhattan Distance heuristic
         between the state of the Node and the final State and the
         current cost of the Node
     * @param finalState Final State of the puzzle
     */
    public void calculateCostUsingHeuristic(State finalState) {
        int distanceFromGoalA = Math.abs(state.getPositionA()[0] -
            finalState.getPositionA()[0]) +
            Math.abs(state.getPositionA()[1] -
            finalState.getPositionA()[1]);
        int distanceFromGoalB = Math.abs(state.getPositionB()[0] -
            finalState.getPositionB()[0]) +
            Math.abs(state.getPositionB()[1] -
            finalState.getPositionB()[1]);
        int distanceFromGoalC = Math.abs(state.getPositionC()[0] -
            finalState.getPositionC()[0]) +
            Math.abs(state.getPositionC()[1] -
            finalState.getPositionC()[1]);
        cost = cost + distanceFromGoalA + distanceFromGoalB +
            distanceFromGoalC;
    }

    /**
     * Returns path from root to a given Node
     * @param nodeToFindPath The Node you want to find the path
         from
     * @return Path from the root to the Node entered as the
         argument
```

```
     */
    public ArrayList<Node> sequence(Node nodeToFindPath) {
        ArrayList<Node> sequence = new ArrayList<>();
        Node node = nodeToFindPath;
        while (node.parent != null) {
            sequence.add(node);
            node = node.parent;
        }
        Collections.reverse(sequence);
        return sequence;
    }

    /**
     * Compares the cost of 2 nodes (used by priority queue for A*)
     */
    @Override
    public int compareTo(Node n) {
        int cost = this.cost - n.getCost();
        //Dealing with nodes that have the same heuristic
        if(cost == 0) {
            if(this.getLevel() > n.getLevel()) {
                return -1;
            } else if (this.getLevel() == n.getLevel()){
                return 0;
            } else {
                return 1;
            }
        } else {
            return cost;
        }
    }
}
```

5.3 Move.java

```
import java.util.Arrays;

/**
```

```
 * Class to do the basic movement of up, down, left and right for
    any given node
 * @author Vedant Chokshi
 */
public class Move {
   /**
    * Do the movement
    * @param n Node upon which the movement needs to be done
    * @param newAgentPosition New position of Agent
    * @return moved Node
    */
   private Node doMovement(Node n, int[] newAgentPosition) {
      Node x = null;
      if(Arrays.equals(n.getState().getPositionA(),
         newAgentPosition)) {
         State s = new State(n.getState().getGridSize(),
            n.getState().getPositionAgent(),
            n.getState().getPositionB(),
            n.getState().getPositionC(), newAgentPosition,
            n.getState().getBlockedTiles());
         x = new Node(s, n, n.getLevel() + 1, n.getCost() + 1);
      } else if(Arrays.equals(n.getState().getPositionB(),
         newAgentPosition)) {
         State s = new State(n.getState().getGridSize(),
            n.getState().getPositionA(),
            n.getState().getPositionAgent(),
            n.getState().getPositionC(), newAgentPosition,
            n.getState().getBlockedTiles());
         x = new Node(s, n, n.getLevel() + 1, n.getCost() + 1);
      } else if(Arrays.equals(n.getState().getPositionC(),
         newAgentPosition)) {
         State s = new State(n.getState().getGridSize(),
            n.getState().getPositionA(),
            n.getState().getPositionB(),
            n.getState().getPositionAgent(), newAgentPosition,
            n.getState().getBlockedTiles());
         x = new Node(s, n, n.getLevel() + 1, n.getCost() + 1);
      } else {
```

```
            State s = new State(n.getState().getGridSize(),
                n.getState().getPositionA(),
                n.getState().getPositionB(),
                n.getState().getPositionC(), newAgentPosition,
                n.getState().getBlockedTiles());
            x = new Node(s, n, n.getLevel() + 1, n.getCost() + 1);
        }
        return x;
    }

    /**
     * Move Left
     * @param n Node upon which the movement needs to be done
     * @return New Node returned as result of the movement
     */
    public Node moveLeft(Node n) {
        if(n.getState().getPositionAgent()[0] - 1 >= 1 &&
            !checkInterference(n.getState().getPositionAgent()[0] -
            1, n.getState().getPositionAgent()[1],
            n.getState().getBlockedTiles())) {
            return doMovement(n, new int[]
                {n.getState().getPositionAgent()[0] - 1,
                n.getState().getPositionAgent()[1]});
        } else {
            return null;
        }
    }

    /**
     * Move Down
     * @param n Node upon which the movement needs to be done
     * @return New Node returned as result of the movement
     */
    public Node moveDown(Node n) {
        if(n.getState().getPositionAgent()[1] + 1 <=
            n.getState().getGridSize()[1] &&
            !checkInterference(n.getState().getPositionAgent()[0],
            n.getState().getPositionAgent()[1] + 1,
```

```
            n.getState().getBlockedTiles())) {
            return doMovement(n, new int[]
               {n.getState().getPositionAgent()[0],
               n.getState().getPositionAgent()[1] + 1});
        } else {
            return null;
        }
    }

    /**
     * Move Right
     * @param n Node upon which the movement needs to be done
     * @return New Node returned as result of the movement
     */
    public Node moveRight(Node n) {
        if(n.getState().getPositionAgent()[0] + 1 <=
            n.getState().getGridSize()[0] &&
            !checkInterference(n.getState().getPositionAgent()[0] +
            1, n.getState().getPositionAgent()[1],
            n.getState().getBlockedTiles())) {
            return doMovement(n, new int[]
               {n.getState().getPositionAgent()[0] + 1,
               n.getState().getPositionAgent()[1]});
        } else {
            return null;
        }
    }

    /**
     * Move Up
     * @param n Node upon which the movement needs to be done
     * @return New Node returned as result of the movement
     */
    public Node moveUp(Node n) {
        if(n.getState().getPositionAgent()[1] - 1 >= 1 &&
            !checkInterference(n.getState().getPositionAgent()[0],
            n.getState().getPositionAgent()[1] - 1,
            n.getState().getBlockedTiles())) {
```

```
            return doMovement(n, new int[]
                {n.getState().getPositionAgent()[0],
                n.getState().getPositionAgent()[1] - 1});
        } else {
            return null;
        }
    }

    /**
     * Checks if a coordinate interferes with the blocked tiles
     * @param x X coordinate
     * @param y Y coordinate
     * @param unreachableCoords List of blocked tiles
     * @return True if there is interference, false other wise
     */
    private boolean checkInterference(int x, int y, int[][]
        unreachableCoords) {
        boolean flag= false;
        if(unreachableCoords != null) {
            for(int[] coord : unreachableCoords) {
                if(x == coord[0] && y == coord[1]) {
                    flag = true;
                    break;
                }
            }
        }
        return flag;
    }
}
```

5.4 Search.java

```
import java.util.*;
/**
 * Search Class contains all the the search method with a main
    method to run the methods
 * @author Vedant Chokshi
 */
```

```
public class Search {

   /**
    * Runs search methods
    */
   public static void main(String[] args) {
      State start = new State(new int[] {4,4}, new int[] {1,4},
         new int[] {2,4}, new int[] {3, 4}, new int[] {4, 4});
      State goal = new State(null, new int[] {2,2}, new int[]
         {2,3}, new int[] {2, 4}, null);
      Node node = new Node(start);
      Search search = new Search();
//    Runs search methods
//    search.BFS(node, goal);
//    search.DFS(node, goal);
//    search.IDS(node, goal);
      search.aStar(node, goal);
   }

   /**
    * Depth-First Search
    * @param root Root Node of the tree which contains the
       initial state of the problem
    * @param goal Final state of the problem (solution)
    */
   public void DFS(Node root, State goal) {
      int nodesExpanded = 0;
      Stack<Node> stack = new Stack<>();
      Move m = new Move();

      System.out.println("Starting Depth-First Search on:\n" +
         root.getState());
      stack.add(root);
      while(!stack.isEmpty()) {
         ArrayList<Node> successors = new ArrayList<>();
         //Pop from top of the stack
         Node current = stack.pop();
         //Check if the solution is found
```

```
            if(checkGoal(current.getState(), goal)) {
                ArrayList<Node> steps = current.sequence(current);
                System.out.println("Finished Depth-First Search with
                    depth - " + current.getCost() + " and nodes
                    expanded - " + nodesExpanded + "\n" +
                    current.getState() + "\nSteps:\n");
                for(Node step : steps) {
                    System.out.println(step.getState());
                }
                break;
            }
            //Expands Node if solution not found
            nodesExpanded++;
            successors.add(m.moveUp(current));
            successors.add(m.moveDown(current));
            successors.add(m.moveLeft(current));
            successors.add(m.moveRight(current));
            //Add successors to the stack (randomly for DFS)
            Collections.shuffle(successors);
            for(Node child : successors) {
                if(child != null) {
                    stack.add(child);
                }
            }
        }
    }

    /**
     * Breadth-First Search
     * @param root Root Node of the tree which contains the
        initial state of the problem
     * @param goal Final state of the problem (solution)
     */
    public void BFS(Node root, State goal) {
        int nodesExpanded = 0;
        Queue<Node> queue = new LinkedList<Node>();
        Move m = new Move();
```

```
        System.out.println("Starting Breadth-First Search on:\n" +
            root.getState());
        queue.add(root);
        while(!queue.isEmpty()) {
            ArrayList<Node> successors = new ArrayList<>();
            //Remove queue head
            Node current = queue.remove();
            //Check if the solution is found
            if(checkGoal(current.getState(), goal)) {
                ArrayList<Node> steps = current.sequence(current);
                System.out.println("Finished Breadth-First Search
                    with depth - " + current.getCost() + " and nodes
                    expanded - " + nodesExpanded + "\n" +
                    current.getState() + "\nSteps:\n");
                for(Node step : steps) {
                    System.out.println(step.getState());
                }
                break;
            }
            //Expands Node if solution not found
            nodesExpanded++;
            successors.add(m.moveUp(current));
            successors.add(m.moveDown(current));
            successors.add(m.moveLeft(current));
            successors.add(m.moveRight(current));
            //Add successors to the queue
            for(Node child : successors) {
                if(child != null) {
                    queue.add(child);
                }
            }
        }
    }

    /**
     * Iterative Deepening Search
     * @param root Root Node of the tree which contains the
        initial state of the problem
```

```
     * @param goal Final state of the problem (solution)
     */
    public void IDS(Node root, State goal) {
        //Initially set maxDepth to 0
        int maxDepth = 0, nodesExpanded = 0;
        Stack<Node> stack = new Stack<>();
        Move m = new Move();

        System.out.println("Starting Iterative Deepening Search
            on:\n" + root.getState());
        stack.add(root);
        while(!stack.isEmpty()) {
            ArrayList<Node> successors = new ArrayList<>();
            //Pop from top of the stack
            Node current = stack.pop();
            //Checks if the popped Node is a solution
            if(checkGoal(current.getState(), goal)) {
                ArrayList<Node> steps = current.sequence(current);
                System.out.println("Finished Iterative Deepening
                    Search with depth - " + current.getCost() + " and
                    nodes expanded - " + nodesExpanded + "\n" +
                    current.getState() + "\nSteps:");
                for(Node step : steps) {
                    System.out.println(step.getState());
                }
                break;
            //Expands Node if the depth of the Node is less than the
                maxDepth
            } else if(current.getLevel() < maxDepth) {
                nodesExpanded++;
                successors.add(m.moveUp(current));
                successors.add(m.moveDown(current));
                successors.add(m.moveLeft(current));
                successors.add(m.moveRight(current));
                for(Node child : successors) {
                    if(child != null) {
                        stack.add(child);
                    }
```

```
            }
        }
        //If stack size = 0, meaning that there is no solution
            for the current maxDepth, increase value of maxDepth
            by 1 and adds the root to the stack so the search can
            be started again with a higher maximum depth
        if(stack.size() == 0) {
            stack.push(root);
            maxDepth++;
        }

    }
}

/**
 * A* Search
 * @param root Root Node of the tree which contains the
    initial state of the problem
 * @param goal Final state of the problem (solution)
 */
public void aStar(Node root, State goal) {
    int nodesExpanded = 0;
    PriorityQueue<Node> queue = new PriorityQueue<>();
    Move m = new Move();

    System.out.println("Starting A* Search on:\n" +
        root.getState());
    queue.add(root);
    while(!queue.isEmpty()) {
        ArrayList<Node> successors = new ArrayList<>();
        //Remove queue head
        Node current = queue.poll();
        //Check if head of the queue is the solution
        if(checkGoal(current.getState(), goal)) {
            ArrayList<Node> steps = current.sequence(current);
            System.out.println("Finished A* Search with depth - "
                + current.getCost() + " and nodes expanded - " +
                nodesExpanded + "\n" + current.getState() +
```

```
                "\nSteps:");
            for(Node step : steps) {
                System.out.println(step.getState());
            }
            break;
        }
        //Expands Node if solution not found
        nodesExpanded++;
        successors.add(m.moveUp(current));
        successors.add(m.moveDown(current));
        successors.add(m.moveLeft(current));
        successors.add(m.moveRight(current));
        //Add successors to the queue
        for(Node child : successors) {
            if(child != null) {
                child.calculateHeuristic(goal);
                queue.add(child);
            }
        }
    }
}

/**
 * Method used to check if final state is achieved
 * @param current State wished to be check
 * @param goal State wished to be check with
 * @return True if current and goal are equal, else return
    false
 */
private boolean checkGoal(State current, State goal) {
    if(Arrays.equals(current.getPositionA(),
        goal.getPositionA()) &&
        Arrays.equals(current.getPositionB(),
        goal.getPositionB()) &&
        Arrays.equals(current.getPositionC(),
        goal.getPositionC())) {
        return true;
    } else {
```

```
            return false;
        }
    }
}
```