Intelligent Systems Assignment:

Search Methods

By Simeon Enchev

Approach

One of the first obstacles I faced was definition of the problem. My first attempt was the most intuitional approach an OOP programmer could take; I tried designing everything as an object. I defined everything as different objects – the grid, the blocks and the ☺. I've also had a class that was designed to entail the state of the current grid configuration.

Definition of Problem

At one point I realized that there was a much simpler and efficient approach that would take less time to implement, less resources to compile and it would be easier for someone else to understand it.

I did the following: I encoded the whole grid and configuration as a **string** in the form of **"000000000000ABCX"**. Obviously, the X is the ☺. Each 0 is an empty block and {A, B, C} are the blocks that we need to build the tower for the goal state (**"00000A000B000C0X"**). Every four symbols form a row in the grid. My whole problem revolves around manipulating the position of the **chars** in the **string.**

Methods

My methods are self-explanatory. The methods that move the **X** are called void **moveLeft(),** void **moveRight(),** void **moveUp(),** void **moveDown().** I have Boolean methods **isMoveRight(), isMoveLeft(), isMoveUp(), isMoveUp()** that check if a movement is possible given the current configuration. The methods for moving **X** also check if movement is possible but this redundant check is there because I needed it for debugging and that doesn't affect performance at all. My other methods are LinkedList<BlockPuzzle> **generateChildBfs,** LinkedList<BlockPuzzle> **generateChildDfs,** ArrayList<BlockPuzzle> **generateChildAStar.** These methods generate children for each algorithm and use different approach in doing it. For **BFS** I check if a move is possible and I add it to the **Linked List**. For **DFS** I have a random number generator that manipulates the order of checks so **DFS** doesn't get stuck. **IDS** uses the same **method** as **DFS**. For **A*** I am doing the same as in **BFS**, however, I am also calculating the **Manhattan distance** and I am adding the **successor** nodes to an **Array List**.

Evidence

To prove that my implementation works, I will provide output for several of the algorithms that showcases the directions needed to find a solution for the given problem. Additionally, I will include information about the nodes expanded, depth and memory used for each algorithm. This should hopefully give a good base idea of how it all works.

BFS Output for reaching Goal State after expanding **8 448 993 nodes**, depth is 14, memory used is around 1.135GB. **BFS** is always consistent, if the configuration doesn't change and the sequence of checks (which move to add first to the queue) remains the same.

0. **[0000 0000 0000 ABCX]** Initial State
1. **[0000 0000 000X ABC0]** UP
2. **[0000 0000 00X0 ABC0]** LEFT
3. **[0000 0000 0X00 ABC0]** LEFT
4. **[0000 0000 0B00 AXC0]** DOWN
5. **[0000 0000 0B00 XAC0]** LEFT
6. **[0000 0000 XB00 0AC0]** UP
7. **[0000 0000 BX00 0AC0]** RIGHT
8. **[0000 0000 BA00 0XC0]** DOWN
9. **[0000 0000 BA00 0CX0]** RIGHT
10. **[0000 0000 BAX0 0C00]** UP
11. **[0000 00X0 BA00 0C00]** UP
12. **[0000 0X00 BA00 0C00]** LEFT
13. **[0000 0A00 BX00 0C00]** DOWN
14. **[0000 0A00 XB00 0C00]** LEFT -> Goal State

DFS Output for reaching Goal State is mostly random in the range of 10 000 – 100 000 nodes expanded. The depth is between 15 000 - 120 000. Memory used is in the range of 0,06GB. I can provide directions for reaching goal state in **DFS** but it going to take a lot of space in the report because of the size of the depth.

IDS Output for reaching Goal State is random in the range of 8 000 000 – 13 000 000 nodes expanded, which is a bit more than **BFS**. In some unlucky cases, it could go above 13 mil. Depth is always 14, just like in **BFS** and memory is around 0,07GB which is similar to **DFS.** For identical configuration, the directions for reaching a goal state are the same as in **BFS**.

A* Output for reaching Goal State is consistent and returns around 1000 nodes expanded for the base problem. The size depends on the sequence of checks (which move to add first to the priority queue). Priority Queue compares **f** for every state (**f = h** (Manhattan distance) + **g** (Depth)) and picks the lower value. Depth is always 14 and memory used is around 0,009GB. I am providing a sequence of moves that is generated by the algorithm and reaches a goal state. The moves are easily trackable and that clearly shows that the algorithm works as intended.

0. **[0000 0000 0000 ABCX]** Initial State
1. **[0000 0000 000X ABC0]** UP
2. **[0000 0000 00X0 ABC0]** LEFT
3. **[0000 0000 0X00 ABC0]** LEFT
4. **[0000 0000 0B00 AXC0]** DOWN
5. **[0000 0000 0B00 XAC0]** LEFT
6. **[0000 0000 XB00 0AC0]** UP

7. **[0000 0000 BX00 0AC0]** RIGHT
8. **[0000 0000 BA00 0XC0]** DOWN
9. **[0000 0000 BA00 0CX0]** RIGHT
10. **[0000 0000 BAX0 0C00]** UP
11. **[0000 00X0 BA00 0C00]** UP
12. **[0000 0X00 BA00 0C00]** LEFT
13. **[0000 0A00 BX00 0C00]** DOWN
14. **[0000 0A00 XB00 0C00]** LEFT -> Goal State

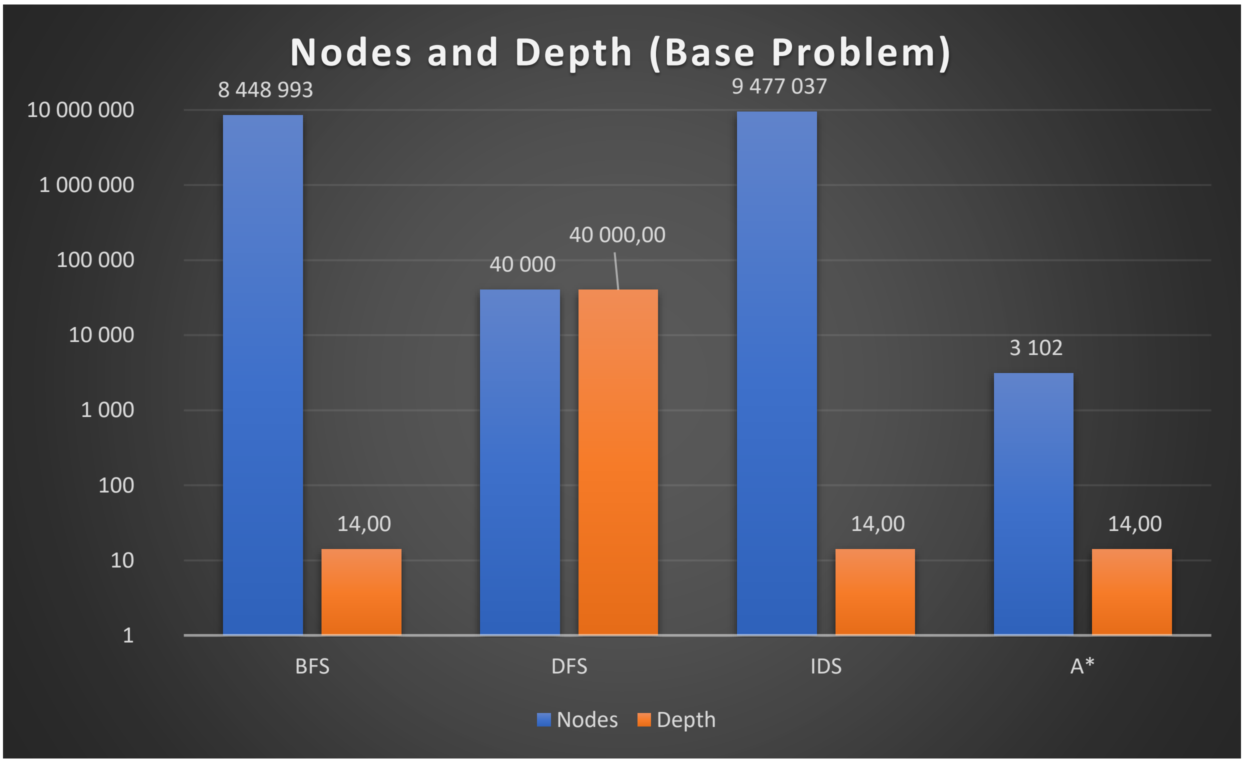
Scalability study

My initial tests were based on the original problem. I've plotted the results on the graphs bellow.

I am using log representation of the values on the graphs.

BFS and **DFS** are the first two algorithms that I have implemented. Both can find a solution, however, there are several distinctive differences between them. The memory graph (in the Extras section) shows that **BFS** takes much more memory (**O(b^d)** space complexity), compared to **DFS** (**O(bd)**). However, **BFS** is much shallower than **DFS** and always finds the optimal solution, which is not the case for **DFS**. That is represented in the difference in the time complexity. **BFS** is **O(b^d)**, while time complexity for **DFS** is **O(b^m),** where **m** is the maximum depth for a node, which is much worse because our tree is infinite (☺ always has at least 2 moves).
From the graphs we can see that **BFS** and **IDS** have the same depth and almost the same number of nodes expanded. However, the main difference is the memory requirement. For **BFS,** we need 1.135GBytes to solve the original problem and for **IDS** that is merely 0,07GBytes which is much closer to the memory value of **DFS.** From the results, we can clearly see that **IDS** combines the best of traits of **BFS** and **DFS** in one; **O(bd)** space complexity of **DFS** and it is optimal and finds the shallowest solution, like **BFS.** Unfortunately, the time complexity for **IDS** is as bad as **BFS, O(b^d),** but it is still better than **DFS.** The reason for this is that it generates all nodes for each level, just like **BFS.** The space complexity is much better because we traverse the nodes as if we are doing **DFS** and we do not store them in memory.
Clearly, **A*** is the best algorithm for this problem. **A*** outperformed every other algorithm in almost every scenario. It finds an optimal solution and time complexity is **O(b^d),** where **d** is solution depth, which is not the worst, compared to the others. Space complexity is not a serious issue because the scale of the problem is small (Graph in Extra section).

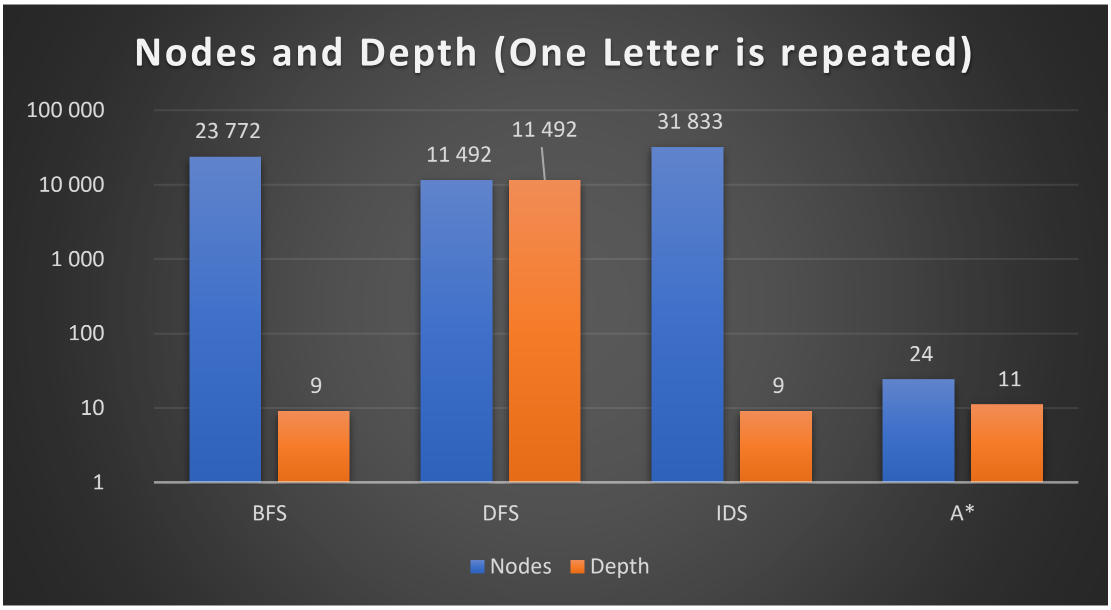
(Values for DFS and IDS are average. DFS is between 10k – 120k and IDS is mostly between 7mil. – 12 mil. I've settled with these values because they are the most common ones.)



I also altered the problem several times to further explore the differences in the characteristics of the algorithms.

```
INPUT "0AAA00000000ABCX"
```

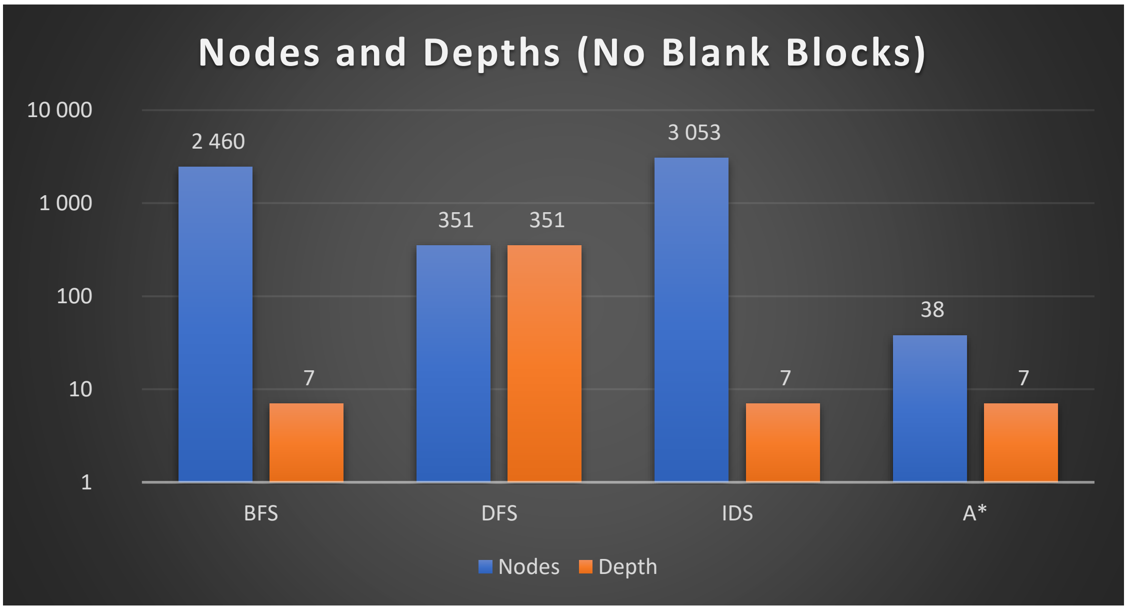
First case makes the problem easier by adding a few more "A" blocks. This significantly boosts the performance for each of the algorithms, especially for A*, which expands less nodes than DFS, however, the solution is not optimal. I tried figuring out why that was the case and I couldn't find anything relevant.



For the second analysis, I am filling all empty blocks with letters. This boosts up the speed even more.

```
INPUT "ABCABCABCABCABCX"
```

In this example, A* clearly outperforms all other algorithms and finds the optimal solution. **BFS** is also doing great because there are not as many states and it even manages to beat **IDS and DFS.** In the next section, the graphs showcase how space complexity of **A*** is the best and how **BFS**'s space complexity is almost the same as **DFS** and **IDS** for this easier problem.



Extras and limitations

Limitations

- Because I am using strings, I can’t directly access the positions of the letter blocks, which could be beneficial in some scenarios.
- Rescaling the problem is also very hard for my implementation and even a 5x5 grid is not solvable for any of my algorithms because it takes way too much memory.
- Also, using strings as nodes just keeps creating new strings in the memory, even though I am using a String Builder, it is still very demanding.
- My DFS children generator method uses 4 different configurations for adding checking for possible moves. A random number from 1 to 4 chooses which configuration is going to be used. I am not sure if that is the best way to randomize the results and, in some cases, it might still cause unnecessary loops, which are not observable while running the code
- My implementation cannot solve the problem, if there are a lot of blank squares in the initial configuration. For example, "00A00000B00X000C" is unsolvable and I run out of memory for every algorithm.

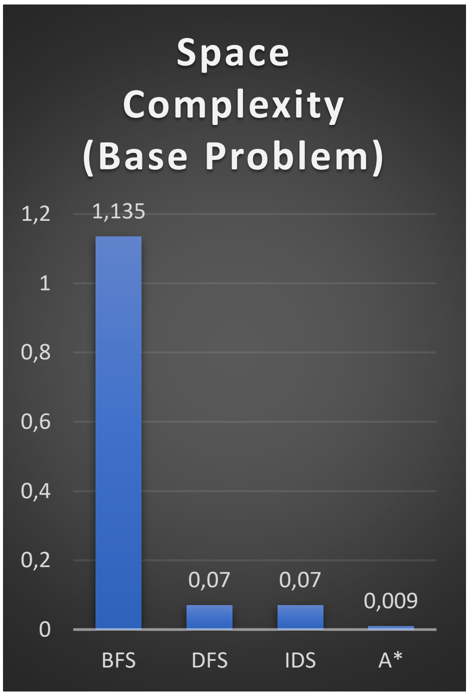
Extras

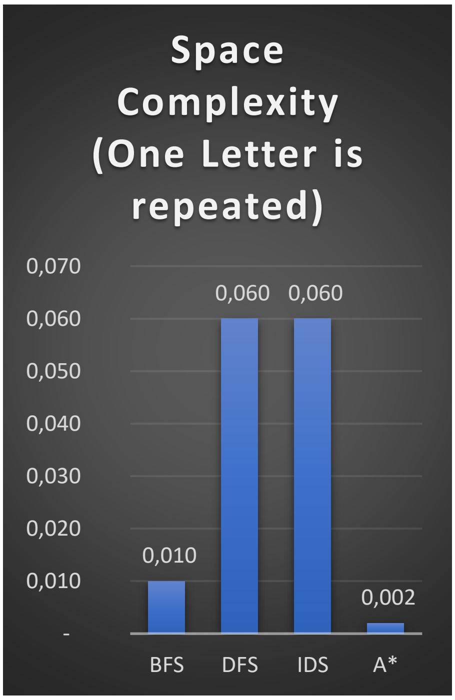
Space complexity for different configurations represented in GBytes.

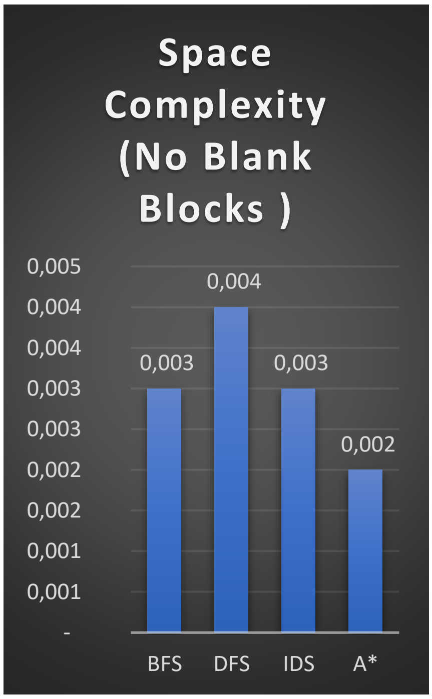
For smaller case problems, space complexity is not an issue, even for **BFS**.

Configurations:

"000000000000ABCX" "0AAA00000000ABCX" "ABCABCABCABCABCX"







We can conclude, that A* is the best algorithm of the four in every case. It has the most efficient time complexity, it expands the least nodes and it requires less memory. Using heuristics is way better than doing uninformed search in terms of efficiency and speed.

References

I've used several webpages for pseudocodes. I've written everything by myself. I've used the pseudocodes only for guidelines.

http://www.mathcs.emory.edu/~cheung/Courses/171/Syllabus/11-Graph/bfs.html

https://en.wikipedia.org/wiki/Depth-first_search

http://www.cs.toronto.edu/~heap/270F02/node36.html

https://www.geeksforgeeks.org/iterative-deepening-searchids-iterative-deepening-depth-first-searchiddfs/

https://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search

Artificial Intelligence, A Modern Approach, Third Edition – Course Book

Code

Main Class

```
import java.util.*;

/**
 * Main class with all algorithms. To test an algorithm, just "comment it
out" and try it.
 * To test another one, comment all other algorithms.
 */
public class Main {

    private static final String goalState = "ABC";
    private static int steps = 0;
    private static BlockPuzzle goalStateIDS; //Variable for the goal state
of IDS, so the directions can be generated.

    public static void main(String[] args){
        long beforeUsedMem=Runtime.getRuntime().totalMemory()-
Runtime.getRuntime().freeMemory(); //Used to calculate memory usage

        BlockPuzzle blockPuzzle = new BlockPuzzle("000000000000ABCX");
//Original Problem
        BlockPuzzle blockPuzzle1 = new BlockPuzzle("0AA0A0000000ABCX");
//Scalability configuration
        BlockPuzzle blockPuzzle2 = new BlockPuzzle("ABCABCABCABCABCX");
//Scalability configuration
        BlockPuzzle blockPuzzle3 = new BlockPuzzle("00A00000B00X000C");
//Unsolvable for my implementation. Runs out of memory.

        /**
         * To test an algorithm, just comment it out and comment all other
algorithms.
         * Modify the arguments for the algorithms if you want to try the
         * different configurations.
         */
        //bfsQUEUE(blockPuzzle);
        //bfsLIST(blockPuzzle);
        //dfs(blockPuzzle);
        //Iterative_Deepening_Search(blockPuzzle);
        //AStar(blockPuzzle);



        /**
         * To test an algorithm and generate directions, comment everything
from the
         * section above and comment out only the algorithm that you want
to see
         * from the section bellow. You can use any of the blockPuzzle
objects
         * to see the results for every configuration.
```

```java
         */
        ArrayList<String> directions = new ArrayList<>();
        directions = generateDirections(bfsListDIRECTIONS(blockPuzzle));
        //directions = generateDirections(bfsQueueDIRECTIONS(blockPuzzle));
        //directions = generateDirections(dfsDIRECTIONS(blockPuzzle));
        //directions =
generateDirections(Iterative_Deepening_SearchDIRECTIONS(blockPuzzle));
        //directions = generateDirections(AStarDIRECTIONS(blockPuzzle));


        for (int i = 0; i < directions.size(); i++) {
            System.out.println(directions.get(i));
        }
        long afterUsedMem=Runtime.getRuntime().totalMemory()-
Runtime.getRuntime().freeMemory();
        System.out.println("Memory used: " + ((afterUsedMem-
beforeUsedMem)/Math.pow(10,9)) + " GBytes");

    }

    public static boolean isGoalState(String currentState){
        return (("" +
currentState.charAt(5)+currentState.charAt(9)+currentState.charAt(13)).equa
ls(goalState));
    }

    /**
     * Explanation for the Methods
     *
     * Methods with DIRECTIONS in the name are used to generate the
directions for
     * finding the solution to each problem. The other methods are there to
just
     * be used for testing if the algorithms work.
     * Either type of the methods use identical logic and the only
difference is
     * that DIRECTION type methods return an Object<BlockPuzzle> and the
other type
     * is void.
     */

    /**
     * For BFS I have two methods because I was not sure what is expected
from us. One of the methods
     * uses a Queue and pops each element before expanding children nodes.
The other methods stores all
     * of the nodes in the Queue and keeps adding new elements without
popping. Even though I am storing the nodes,
     * I am not checking for repetitions in the states.
     */
    public static void bfsQUEUE(BlockPuzzle initialState){
        LinkedList<BlockPuzzle> queue = new LinkedList<BlockPuzzle>();
        queue.add(initialState);

        int steps = 0;

        while (queue.size() != 0){
            BlockPuzzle state = queue.poll();

            Iterator<BlockPuzzle> i =
BlockPuzzle.generateChildBfs(state).listIterator();
```

```java
            while (i.hasNext()){

                BlockPuzzle node = i.next();

                if (!(isGoalState(node.getCurrentState()))){
                    queue.add(node);
                    steps++;

                } else {
                    System.out.println();
                    System.out.println("Steps: " + steps);
                    System.out.println("Goal at depth: " +
node.getStateDepth());
                    System.out.println("Queue size: " + queue.size());
                    System.out.println("Goal state: " +
node.getCurrentState());
                    queue.clear();
                    break;

                }
            }
        }

    }

    public static void bfsLIST(BlockPuzzle initialState){
        ArrayList<BlockPuzzle> queue = new ArrayList<>();
        queue.add(initialState);
        int counter = 0;
        int steps = 0;

        while (queue.size() != 0){
            BlockPuzzle state = queue.get(counter);

            Iterator<BlockPuzzle> i =
BlockPuzzle.generateChildBfs(state).listIterator();

            while (i.hasNext()){

                BlockPuzzle node = i.next();

                if (!(isGoalState(node.getCurrentState()))){
                    queue.add(node);
                    steps++;

                } else {
                    System.out.println();
                    System.out.println("Steps: " + steps);
                    System.out.println("Goal at depth: " +
node.getStateDepth());
                    System.out.println("Queue size: " + queue.size());
                    System.out.println("Goal state: " +
node.getCurrentState());
                    queue.clear();
                    break;

                }
            }
            counter++;
        }
```

```java
    }

    public static BlockPuzzle bfsQueueDIRECTIONS(BlockPuzzle initialState){
        LinkedList<BlockPuzzle> queue = new LinkedList<BlockPuzzle>();
        queue.add(initialState);

        int steps = 0;

        while (queue.size() != 0){
            BlockPuzzle state = queue.poll();

            Iterator<BlockPuzzle> i =
BlockPuzzle.generateChildBfs(state).listIterator();

            while (i.hasNext()){

                BlockPuzzle node = i.next();

                if (!(isGoalState(node.getCurrentState()))){
                    queue.add(node);
                    steps++;

                } else {
                    System.out.println();
                    System.out.println("Steps: " + steps);
                    System.out.println("Goal at depth: " +
node.getStateDepth());
                    System.out.println("Queue size: " + queue.size());
                    System.out.println("Goal state: " +
node.getCurrentState());
                    queue.clear();
                    return node;

                }
            }
        }
        return initialState;

    }

    public static BlockPuzzle bfsListDIRECTIONS(BlockPuzzle initialState){
        ArrayList<BlockPuzzle> queue = new ArrayList<>();
        queue.add(initialState);
        int counter = 0;
        int steps = 0;

        while (queue.size() != 0){
            BlockPuzzle state = queue.get(counter);

            Iterator<BlockPuzzle> i =
BlockPuzzle.generateChildBfs(state).listIterator();

            while (i.hasNext()){

                BlockPuzzle node = i.next();

                if (!(isGoalState(node.getCurrentState()))){
                    queue.add(node);
                    steps++;
```

```
                } else {
                    System.out.println();
                    System.out.println("Steps: " + steps);
                    System.out.println("Goal at depth: " +
node.getStateDepth());
                    System.out.println("Queue size: " + queue.size());
                    System.out.println("Goal state: " +
node.getCurrentState());
                    queue.clear();
                    return node;

                }
            }
            counter++;
        }
        return initialState;

    }

    public static void dfs(BlockPuzzle initialState){
        Stack<BlockPuzzle> stack = new Stack<>();
        stack.push(initialState);


        ArrayList<BlockPuzzle> children;

        int steps = 0;

        while (stack.size() != 0){
            BlockPuzzle parent = stack.pop();
            steps++;

            if ((isGoalState(parent.getCurrentState()))){
                stack.push(parent);
                System.out.println("Steps: " + steps);
                System.out.println("Goal at depth: " +
parent.getStateDepth());
                System.out.println("Goal state: " +
parent.getCurrentState());
                stack.clear();
                break;
            } else {
                for (int i = 0; i <
BlockPuzzle.generateChildDfsLIST(parent).size(); i++) {
                    children = BlockPuzzle.generateChildDfsLIST(parent);
                    stack.push(children.get(i));
                }

            }

        }

    }

    public static BlockPuzzle dfsDIRECTIONS(BlockPuzzle initialState){
        Stack<BlockPuzzle> stack = new Stack<>();
        stack.push(initialState);

        int steps = 0;

        while (stack.size() != 0){
```

```
            BlockPuzzle state = stack.pop();

            Iterator<BlockPuzzle> i =
BlockPuzzle.generateChildDfs(state).listIterator();

            while (i.hasNext()){
                BlockPuzzle node = i.next();
                if (!(isGoalState(node.getCurrentState()))){
                    stack.push(node);
                    System.out.println(node.getCurrentState());
                    steps++;

                } else {
                    System.out.println();
                    System.out.println("Steps: " + steps);
                    System.out.println("Goal at depth: " +
node.getStateDepth());
                    System.out.println("Goal state: " +
node.getCurrentState());
                    stack.clear();
                    return node;

                }

            }

        }
        return initialState;
    }

    public static boolean Depth_Limited_Search(BlockPuzzle problem, int
limit) {
        Stack<BlockPuzzle> stack = new Stack<>();
        stack.push(problem);

        boolean cut_off = false;

        while (stack.size() != 0) {
            BlockPuzzle parent = stack.pop();
            if (isGoalState(parent.getCurrentState())) {
                System.out.println();
                System.out.println("Steps: " + steps);
                System.out.println("Goal at depth: " +
parent.getStateDepth());
                System.out.println("Goal state: " +
parent.getCurrentState());
                goalStateIDS = parent;
                cut_off = true;
                break;

            }

            if (parent.getStateDepth() == limit) {
                continue;
            } else {
                for (BlockPuzzle i : BlockPuzzle.generateChildDfs(parent))
{
                    stack.push(i);
                    steps++;

                }
```

```
            }
        }
        return cut_off;
    }

    public static void Iterative_Deepening_Search(BlockPuzzle problem){
        boolean cut_off = false;
        int depth = 0;

        while (!cut_off){
            cut_off = Depth_Limited_Search(problem, depth);
            depth+=1;
        }
    }

    public static BlockPuzzle
Iterative_Deepening_SearchDIRECTIONS(BlockPuzzle problem){
        boolean cut_off = false;
        int depth = 0;

        while (!cut_off){
            cut_off = Depth_Limited_Search(problem, depth);
            depth+=1;
        }

        return goalStateIDS;
    }

    public static void AStar(BlockPuzzle problem){

        PriorityQueue<BlockPuzzle> pQueue = new
PriorityQueue<BlockPuzzle>();
        ArrayList<BlockPuzzle> children;

        pQueue.add(problem);
        int step = 0;

        while (pQueue.size() != 0){
            BlockPuzzle parent = pQueue.poll();
            step++;

            if ((isGoalState(parent.getCurrentState()))){
                System.out.println();
                System.out.println("Steps: " + step);
                System.out.println("Goal at depth: " +
parent.getStateDepth());
                System.out.println("Goal state: " +
parent.getCurrentState());
                pQueue.clear();
                break;

            } else {
                for (int i = 0; i <
BlockPuzzle.generateChildAStarList(parent).size(); i++) {
                    children = BlockPuzzle.generateChildAStarList(parent);
                    pQueue.add(children.get(i));

                }
            }

        }
```

```
    }

    public static BlockPuzzle AStarDIRECTIONS(BlockPuzzle problem){

        PriorityQueue<BlockPuzzle> pQueue = new
PriorityQueue<BlockPuzzle>();
        ArrayList<BlockPuzzle> children;

        pQueue.add(problem);
        int step = 0;

        while (pQueue.size() != 0){
            BlockPuzzle parent = pQueue.poll();
            step++;
            if ((isGoalState(parent.getCurrentState()))){
                System.out.println();
                System.out.println("Steps: " + step);
                System.out.println("Goal at depth: " +
parent.getStateDepth());
                System.out.println("Goal state: " +
parent.getCurrentState());
                pQueue.clear();
                return parent;

            } else {
                for (int i = 0; i <
BlockPuzzle.generateChildAStarList(parent).size(); i++) {
                    children = BlockPuzzle.generateChildAStarList(parent);
                    pQueue.add(children.get(i));
                }
            }

        }
        return problem;
    }

    /**
     * This methods generates the directions for each of the methods by
filling an ArrayList
     * with the state of each predecessor of the goal state.
     * @param problem the goal state which will be used to generate the
directions
     * @return ArrayList with directions for finding the solution
     */
    public static ArrayList<String> generateDirections(BlockPuzzle
problem){
        ArrayList<String> directions = new ArrayList<>();
        String temp = problem.getCurrentState() + " " +
problem.getDirection();
        directions.add(temp);

        while (problem.getParent() != problem){
            directions.add(problem.getParent().getCurrentState() + " " +
problem.getParent().getDirection());
            problem = problem.getParent();
        }

        directions.remove(directions.size()-1);
        directions.add(problem.getParent().getCurrentState() + " " +
"Initial State");
```

```
        Collections.reverse(directions);
        return directions;

    }
}
```

BlockPuzzle class

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

/**
 * This class defines the 8-puzzle problem given in the coursework
specifications. I've decided to
 * use a string to represent the problem.
 *
 * EXAMPLE: 000000000000ABCX is the initial configuration and represents
 * 0 0 0 0   X.(moveLeft)   0 0 0 0
 * 0 0 0 0   ------------>  0 0 0 0 =====> 000000000000ABXC
 * 0 0 0 0                  0 0 0 0
 * A B C(X)                 A B(X)C
 *      where,  X is the smiley face that moves the other tiles
 *              0 is an empty block
 *              {A,B,C} are the blocks that have to be moved
 *
 * The string represents the grid and is efficient and fast. I've found
this approach to be
 * the fastest because it is not too complicated, it is memory efficient
and easy to implement.
 */

public class BlockPuzzle implements Comparable<BlockPuzzle> {

    private static final String goalState = "ABC"; //Used for checking the
goal state
    private String currentState;
    private String initialState;
    private Integer size = 16; //Size of the grid
    private Integer smileyPosition; //Position of the smiley face
    private static Integer treeDepth = 0;
    private Integer stateDepth = 0;
    private BlockPuzzle parent;
    private static final char[] symbols = {'A','B','C','X'}; //Symbols in
the system that are being moved
    private Integer g; //Expanded nodes for Manhattan Distance
    private Integer f; //Manhattan distance
    private final static String[] directions = {"LEFT", "RIGHT", "UP",
"DOWN"};
    private String direction;

    /**
     * Simple constructor that builds the grid for the original problem.
     */
    public BlockPuzzle(){
        StringBuilder stringBuilder = new StringBuilder();
        int n = 0;
```

```java
        for (int i = 0; i < size; i++) {
            if (i <= 11){ //was 11
                stringBuilder.append("0");
            } else {
                stringBuilder.append(symbols[n]);
                n++;
            }
        }
        initialState = stringBuilder.toString();
        smileyPosition = 15; //was 15
        currentState = initialState;
        parent = this;
        direction = null;
        g = 0;
    }

    /**
     * Constructor for custom configurations.
     * @param initialState the configuration for the problem
     */
    public BlockPuzzle(String initialState){
        smileyPosition = 15; //was 15
        currentState = initialState;
        parent = this;
        g = 0;
    }

    /**
     * Constructor for building a grid with different size.
     * Redundant and not used because 5x5 is unsolvable for my
implementation.
     * @param size
     */
    public BlockPuzzle(int size){
        StringBuilder stringBuilder = new StringBuilder();
        this.size = size*size;

        int n = 0;

        for (int i = 0; i < this.size; i++) {
            if (i <= this.size-5){
                stringBuilder.append("0");
            } else {
                stringBuilder.append(symbols[n]);
                n++;
            }
        }
        initialState = stringBuilder.toString();
        smileyPosition = this.size - 1;
        currentState = initialState;
        parent = this;
        g = 0;
    }


    /**
     *
     *
     * Getters and setters
     *
     *
```

```java
     *
     */

    public static String getGoalState() {
        return goalState;
    }

    public String getCurrentState() {
        return currentState;
    }

    public void setCurrentState(String currentState) {
        this.currentState = currentState;
    }

    public String getInitialState() {
        return initialState;
    }

    public void setInitialState(String initialState) {
        this.initialState = initialState;
    }

    public Integer getSize() {
        return size;
    }

    public void setSize(Integer size) {
        this.size = size;
    }

    public Integer getSmileyPosition() {
        return smileyPosition;
    }

    public void setSmileyPosition(Integer smileyPosition) {
        this.smileyPosition = smileyPosition;
    }

    public static Integer getTreeDepth() {
        return treeDepth;
    }

    public static void setTreeDepth(Integer treeDepth) {
        BlockPuzzle.treeDepth = treeDepth;
    }

    public Integer getStateDepth() {
        return stateDepth;
    }

    public void setStateDepth(Integer stateDepth) {
        this.stateDepth = stateDepth;
    }

    public BlockPuzzle getParent() {
        return parent;
    }

    public void setParent(BlockPuzzle parent) {
        this.parent = parent;
```

```
    }

    public Integer getG() {
        return g;
    }

    public void setG(Integer g) {
        this.g = g;
    }

    public Integer getF() {
        return f;
    }

    public void setF(Integer f) {
        this.f = f;
    }

    public String getDirection() {
        return direction;
    }

    public void setDirection(String direction) {
        this.direction = direction;
    }

    /**
     *
     *
     * Class methods
     *
     *
     */

    /**
     * Moves the smiley face to the left
     * Checks if move is possible
     * If possible ->   If position is occupied by a letter, swap smiley
and letter
     *                  If not, move smiley and empty previous smiley
position
     * Else -> Don't do anything
     * @return new state with smiley to the left
     */
    private String moveLeft(){
        //if (smileyPosition > 0)
        if (((smileyPosition > 0 && smileyPosition <= 3) || (smileyPosition
> 4 && smileyPosition <= 7)
                || (smileyPosition > 8 && smileyPosition <= 11) ||
(smileyPosition > 12 && smileyPosition <=15)))
        {  //Checks if move to the left is possible
            smileyPosition--;
            //Checks if there is a letter in the next box, if there is
            //swap positions of letter and smiley
            //if not, just move smiley and leave previous smiley position
blank
            if (currentState.charAt(smileyPosition) != '0'){
                char dummy = currentState.charAt(smileyPosition);
                char[] tempState = currentState.toCharArray();
                tempState[smileyPosition] = 'X';
                tempState[smileyPosition+1] = dummy;
```

```
                currentState = new String(tempState);
                return currentState;
            } else {
                char[] tempState = currentState.toCharArray();
                tempState[smileyPosition] = 'X';
                tempState[smileyPosition+1] = '0';
                currentState = new String(tempState);
                return currentState;
            }
        } else {
            return currentState;
        }

    }

    /**
     * Moves the smiley face to the right
     * Checks if move is possible
     * If possible ->   If position is occupied by a letter, swap smiley
and letter
     *                  If not, move smiley and empty previous smiley
position
     * Else -> Don't do anything
     * @return new state with smiley to the right
     */
    private String moveRight(){
        //if (smileyPosition < 15)
        //if (smileyPosition < size-1)
        if ((smileyPosition >= 0 && smileyPosition < 3) || (smileyPosition
>= 4 && smileyPosition < 7)
                || (smileyPosition >= 8 && smileyPosition < 11) ||
(smileyPosition >= 12 && smileyPosition < 15))
        {
            smileyPosition++;
            if (currentState.charAt(smileyPosition) != '0'){
                char dummy = currentState.charAt(smileyPosition);
                char[] tempState = currentState.toCharArray();
                tempState[smileyPosition] = 'X';
                tempState[smileyPosition-1] = dummy;
                currentState = new String(tempState);
                return currentState;
            } else {
                char[] tempState = currentState.toCharArray();
                tempState[smileyPosition] = 'X';
                tempState[smileyPosition-1] = '0';
                currentState = new String(tempState);
                return currentState;
            }
        } else {
            return currentState;
        }
    }

    /**
     * Moves the smiley face up
     * Checks if move is possible
     * If possible ->   If position is occupied by a letter, swap smiley
and letter
     *                  If not, move smiley and empty previous smiley
position
     * Else -> Don't do anything
```

```
     * @return new state with smiley moved up
     */
    private String moveUp(){
        //if (smileyPosition <= 15 && smileyPosition >= 4)
        if (smileyPosition <= size-1 && smileyPosition >= Math.sqrt(size)){
            smileyPosition-=4;
            if (currentState.charAt(smileyPosition) != '0'){
                char dummy = currentState.charAt(smileyPosition);
                char[] tempState = currentState.toCharArray();
                tempState[smileyPosition] = 'X';
                tempState[smileyPosition+4] = dummy;
                currentState = new String(tempState);
                return currentState;
            } else {
                char[] tempState = currentState.toCharArray();
                tempState[smileyPosition] = 'X';
                tempState[smileyPosition+4] = '0';
                currentState = new String(tempState);
                return currentState;
            }
        } else {
            return currentState;
        }
    }

    /**
     * Moves the smiley face down
     * Checks if move is possible
     * If possible ->   If position is occupied by a letter, swap smiley
and letter
     *                  If not, move smiley and empty previous smiley
position
     * Else -> Don't do anything
     * @return new state with smiley moved down
     */
    private String moveDown(){
        //if (smileyPosition <= 11 && smileyPosition >= 0)
        if (smileyPosition <= (size-1)-Math.sqrt(size) && smileyPosition >=
0){
            smileyPosition+=4;
            if (currentState.charAt(smileyPosition) != '0'){
                char dummy = currentState.charAt(smileyPosition);
                char[] tempState = currentState.toCharArray();
                tempState[smileyPosition] = 'X';
                tempState[smileyPosition-4] = dummy;
                currentState = new String(tempState);
                return currentState;
            } else {
                char[] tempState = currentState.toCharArray();
                tempState[smileyPosition] = 'X';
                tempState[smileyPosition-4] = '0';
                currentState = new String(tempState);
                return currentState;
            }
        } else {
            return currentState;
        }
    }

    /**
     *
```

```java
     * Dummy methods that I've used for tests.
     *
     */
    private String dummyLeft(){
        Integer smileyPosition = getSmileyPosition();
        String currentState = getCurrentState();
        if (smileyPosition > 0){
            smileyPosition--;
            if (currentState.charAt(smileyPosition) != '0'){
                char dummy = currentState.charAt(smileyPosition);
                char[] tempState = currentState.toCharArray();
                tempState[smileyPosition] = 'X';
                tempState[smileyPosition+1] = dummy;
                currentState = new String(tempState);
                return currentState;
            } else {
                char[] tempState = currentState.toCharArray();
                tempState[smileyPosition] = 'X';
                tempState[smileyPosition+1] = '0';
                currentState = new String(tempState);
                return currentState;
            }
        } else {
            return currentState;
        }
    }

    private String dummyRight(){
        Integer smileyPosition = getSmileyPosition();
        String currentState = getCurrentState();

        if (smileyPosition < 15){
            smileyPosition++;
            if (currentState.charAt(smileyPosition) != '0'){
                char dummy = currentState.charAt(smileyPosition);
                char[] tempState = currentState.toCharArray();
                tempState[smileyPosition] = 'X';
                tempState[smileyPosition-1] = dummy;
                currentState = new String(tempState);
                return currentState;
            } else {
                char[] tempState = currentState.toCharArray();
                tempState[smileyPosition] = 'X';
                tempState[smileyPosition-1] = '0';
                currentState = new String(tempState);
                return currentState;
            }
        } else {
            return currentState;
        }
    }

    private String dummyUp(){
        Integer smileyPosition = getSmileyPosition();
        String currentState = getCurrentState();

        if (smileyPosition <= 15 && smileyPosition >= 4){
            smileyPosition-=4;
            if (currentState.charAt(smileyPosition) != '0'){
                char dummy = currentState.charAt(smileyPosition);
                char[] tempState = currentState.toCharArray();
```

```java
                tempState[smileyPosition] = 'X';
                tempState[smileyPosition+4] = dummy;
                currentState = new String(tempState);
                return currentState;
            } else {
                char[] tempState = currentState.toCharArray();
                tempState[smileyPosition] = 'X';
                tempState[smileyPosition+4] = '0';
                currentState = new String(tempState);
                return currentState;
            }
        } else {
            return currentState;
        }
    }

    private String dummyDown(){
        Integer smileyPosition = getSmileyPosition();
        String currentState = getCurrentState();

        if (smileyPosition <= 11 && smileyPosition >= 0){
            smileyPosition+=4;
            if (currentState.charAt(smileyPosition) != '0'){
                char dummy = currentState.charAt(smileyPosition);
                char[] tempState = currentState.toCharArray();
                tempState[smileyPosition] = 'X';
                tempState[smileyPosition-4] = dummy;
                currentState = new String(tempState);
                return currentState;
            } else {
                char[] tempState = currentState.toCharArray();
                tempState[smileyPosition] = 'X';
                tempState[smileyPosition-4] = '0';
                currentState = new String(tempState);
                return currentState;
            }
        } else {
            return currentState;
        }
    }

    /**
     *
     *
     * Children generators for all methods
     *
     *
     */

    /**
     * Generates children for each state for the problem
     * @param parent of the children that are going to be generated
     * @return LinkedList with the children states
     */
    public static LinkedList<BlockPuzzle> generateChildBfs(BlockPuzzle
parent){
        LinkedList<BlockPuzzle> childStates = new LinkedList<>();
        if (parent.isMovementPossible()){
            if (parent.isMoveRight()){
                BlockPuzzle child2 = new BlockPuzzle();
                child2.smileyPosition = parent.getSmileyPosition();
```

```java
                child2.initialState = parent.getCurrentState();
                child2.currentState = child2.initialState;
                child2.stateDepth = parent.stateDepth+1;
                child2.parent = parent;
                child2.direction = directions[1];
                child2.moveRight();
                childStates.add(child2);
            }

            if (parent.isMoveDown()){
                BlockPuzzle child4 = new BlockPuzzle();
                child4.smileyPosition = parent.getSmileyPosition();
                child4.initialState = parent.getCurrentState();
                child4.currentState = child4.initialState;
                child4.stateDepth = parent.stateDepth+1;
                child4.parent = parent;
                child4.direction = directions[3];
                child4.moveDown();
                childStates.add(child4);
            }
            if (parent.isMoveLeft()){
                BlockPuzzle child1 = new BlockPuzzle();
                if (treeDepth < 1){
                    parent.direction = directions[0];
                }
                child1.smileyPosition = parent.getSmileyPosition();
                child1.initialState = parent.getCurrentState();
                child1.stateDepth = parent.stateDepth+1;
                child1.currentState = child1.initialState;
                child1.parent = parent;
                child1.direction = directions[0];
                child1.moveLeft();
                childStates.add(child1);
            }

            if (parent.isMoveUp()){
                if (treeDepth < 1){
                    parent.direction = directions[2];
                }
                BlockPuzzle child3 = new BlockPuzzle();
                child3.smileyPosition = parent.getSmileyPosition();
                child3.initialState = parent.getCurrentState();
                child3.currentState = child3.initialState;
                child3.stateDepth = parent.stateDepth+1;
                child3.parent = parent;
                child3.direction = directions[2];
                child3.moveUp();
                childStates.add(child3);
            }


        }
        treeDepth = parent.stateDepth+1;
        return childStates;
    }

    /**
     * Generates children for each state of the problem. The order is
randomized because
     * DFS tends to get stuck.
     * @param parent of the children that are going to be generated
```

```java
     * @return LinkedList with the children states
     */
    public static LinkedList<BlockPuzzle> generateChildDfs(BlockPuzzle
parent){
        LinkedList<BlockPuzzle> childStates = new LinkedList<>();

        if (parent.isMovementPossible()) {
            if (parent.isMoveLeft()){
                BlockPuzzle child1 = new BlockPuzzle();
                if (treeDepth < 1){
                    parent.direction = directions[0];
                }
                child1.smileyPosition = parent.getSmileyPosition();
                child1.initialState = parent.getCurrentState();
                child1.currentState = child1.initialState;
                child1.parent = parent;
                child1.stateDepth = parent.stateDepth+1;
                child1.direction = directions[0];
                child1.moveLeft();
                childStates.add(child1);
            }
            if (parent.isMoveDown()){
                BlockPuzzle child4 = new BlockPuzzle();
                child4.smileyPosition = parent.getSmileyPosition();
                child4.initialState = parent.getCurrentState();
                child4.currentState = child4.initialState;
                child4.parent = parent;
                child4.stateDepth = parent.stateDepth+1;
                child4.direction = directions[3];
                child4.moveDown();
                childStates.add(child4);
            }
            if (parent.isMoveRight()){
                BlockPuzzle child2 = new BlockPuzzle();
                if (treeDepth < 1){
                    parent.direction = directions[1];
                }
                child2.smileyPosition = parent.getSmileyPosition();
                child2.initialState = parent.getCurrentState();
                child2.currentState = child2.initialState;
                child2.parent = parent;
                child2.stateDepth = parent.stateDepth+1;
                child2.direction = directions[1];
                child2.moveRight();
                childStates.add(child2);
            }
            if (parent.isMoveUp()){
                BlockPuzzle child3 = new BlockPuzzle();
                if (treeDepth < 1){
                    parent.direction = directions[2];
                }
                child3.smileyPosition = parent.getSmileyPosition();
                child3.initialState = parent.getCurrentState();
                child3.currentState = child3.initialState;
                child3.parent = parent;
                child3.stateDepth = parent.stateDepth+1;
                child3.direction = directions[2];
                child3.moveUp();
                childStates.add(child3);
            }
```

```java
        }
        treeDepth = parent.stateDepth + 1;
        Collections.shuffle(childStates);
        return childStates;
    }


    public static ArrayList<BlockPuzzle> generateChildDfsLIST(BlockPuzzle
parent){
        //Random random = new Random();
        //int n = random.nextInt(3)+1;
        ArrayList<BlockPuzzle> childStates = new ArrayList<>();
        //int position = parent.smileyPosition;

        if (parent.isMovementPossible()){
            if (parent.isMoveRight()){
                BlockPuzzle child2 = new BlockPuzzle();
                child2.smileyPosition = parent.getSmileyPosition();
                child2.initialState = parent.getCurrentState();
                child2.currentState = child2.initialState;
                child2.stateDepth = parent.stateDepth+1;
                child2.parent = parent;
                child2.direction = directions[1];
                child2.moveRight();
                childStates.add(child2);
            }

            if (parent.isMoveDown()){
                BlockPuzzle child4 = new BlockPuzzle();
                child4.smileyPosition = parent.getSmileyPosition();
                child4.initialState = parent.getCurrentState();
                child4.currentState = child4.initialState;
                child4.stateDepth = parent.stateDepth+1;
                child4.parent = parent;
                child4.direction = directions[3];
                child4.moveDown();
                childStates.add(child4);
            }
            if (parent.isMoveLeft()){
                BlockPuzzle child1 = new BlockPuzzle();
                if (treeDepth < 1){
                    parent.direction = directions[0];
                }
                child1.smileyPosition = parent.getSmileyPosition();
                child1.initialState = parent.getCurrentState();
                child1.stateDepth = parent.stateDepth+1;
                child1.currentState = child1.initialState;
                child1.parent = parent;
                child1.direction = directions[0];
                child1.moveLeft();
                childStates.add(child1);
            }

            if (parent.isMoveUp()){
                if (treeDepth < 1){
                    parent.direction = directions[2];
                }
                BlockPuzzle child3 = new BlockPuzzle();
                child3.smileyPosition = parent.getSmileyPosition();
                child3.initialState = parent.getCurrentState();
                child3.currentState = child3.initialState;
```

```java
                child3.stateDepth = parent.stateDepth+1;
                child3.parent = parent;
                child3.direction = directions[2];
                child3.moveUp();
                childStates.add(child3);
            }
        }
        treeDepth = parent.stateDepth + 1;
        Collections.shuffle(childStates);
        return childStates;
    }

    /**
     * Generates children for each state of the problem.  Not used in the
implementation.
     * @param parent of the children that are going to be generated
     * @return LinkedList with the children states
     */
    public static LinkedList<BlockPuzzle> generateChildAStar(BlockPuzzle
parent){
        LinkedList<BlockPuzzle> childStates = new LinkedList<>();

        if (parent.isMovementPossible()){
            if (parent.isMoveRight()){
                BlockPuzzle child2 = new BlockPuzzle();
                child2.smileyPosition = parent.getSmileyPosition();
                child2.initialState = parent.getCurrentState();
                child2.currentState = child2.initialState;
                child2.stateDepth = parent.stateDepth+1;
                child2.parent = parent;
                child2.g = parent.g + 1;
                child2.moveRight();
                child2.f = manhattanDistance(child2.getCurrentState()) +
child2.g;
                childStates.add(child2);
            }

            if (parent.isMoveLeft()){
                BlockPuzzle child1 = new BlockPuzzle();
                child1.smileyPosition = parent.getSmileyPosition();
                child1.initialState = parent.getCurrentState();
                child1.stateDepth = parent.stateDepth+1;
                child1.currentState = child1.initialState;
                child1.parent = parent;
                child1.moveLeft();
                child1.g = parent.g + 1;
                child1.moveRight();
                child1.f = manhattanDistance(child1.getCurrentState()) +
child1.g;
                childStates.add(child1);
            }
            if (parent.isMoveUp()){
                BlockPuzzle child3 = new BlockPuzzle();
                child3.smileyPosition = parent.getSmileyPosition();
                child3.initialState = parent.getCurrentState();
                child3.currentState = child3.initialState;
                child3.stateDepth = parent.stateDepth+1;
                child3.parent = parent;
                child3.moveUp();
                child3.g = parent.g + 1;
                child3.moveRight();
```

```java
                child3.f = manhattanDistance(child3.getCurrentState()) +
child3.g;
                childStates.add(child3);
            }
            if (parent.isMoveDown()){
                BlockPuzzle child4 = new BlockPuzzle();
                child4.smileyPosition = parent.getSmileyPosition();
                child4.initialState = parent.getCurrentState();
                child4.currentState = child4.initialState;
                child4.stateDepth = parent.stateDepth+1;
                child4.parent = parent;
                child4.moveDown();
                child4.g = parent.g + 1;
                child4.moveRight();
                child4.f = manhattanDistance(child4.getCurrentState()) +
child4.g;
                childStates.add(child4);
            }
        }
        treeDepth = parent.stateDepth+1;
        return childStates;
    }

    /**
     * Generates children for each state of the problem. Takes into account
the Manhattan distance
     * to the goal and sets the priority for the priority queue
     * @param parent of the children that are going to be generated
     * @return LinkedList with the children states
     */
    public static ArrayList<BlockPuzzle> generateChildAStarList(BlockPuzzle
parent){
        ArrayList<BlockPuzzle> childStates = new ArrayList<>();

        if (parent.isMovementPossible()){
            if (parent.isMoveLeft()){
                BlockPuzzle child1 = new BlockPuzzle();
                //System.out.println("Left");
                if (treeDepth < 1){
                    parent.direction = directions[0];
                }
                child1.smileyPosition = parent.getSmileyPosition();
                child1.initialState = parent.getCurrentState();
                child1.stateDepth = parent.stateDepth+1;
                child1.currentState = child1.initialState;
                child1.parent = parent;
                child1.direction = directions[0];
                child1.moveLeft();
                child1.g = parent.g + 1;
                //child1.moveRight();
                child1.f = manhattanDistance(child1.getCurrentState()) +
child1.g;
                childStates.add(child1);
            }
            if (parent.isMoveUp()){
                BlockPuzzle child3 = new BlockPuzzle();
                //System.out.println("Up");
                if (treeDepth < 1){
                    parent.direction = directions[2];
                }
                child3.smileyPosition = parent.getSmileyPosition();
```

```
                child3.initialState = parent.getCurrentState();
                child3.currentState = child3.initialState;
                child3.stateDepth = parent.stateDepth+1;
                child3.parent = parent;
                child3.direction = directions[2];
                child3.moveUp();
                child3.g = parent.g + 1;
                child3.f = manhattanDistance(child3.getCurrentState()) +
child3.g;
                childStates.add(child3);
            }
            if (parent.isMoveRight()){
                BlockPuzzle child2 = new BlockPuzzle();
                //System.out.println("Right");
                if (treeDepth < 1){
                    parent.direction = directions[1];
                }
                child2.smileyPosition = parent.getSmileyPosition();
                child2.initialState = parent.getCurrentState();
                child2.currentState = child2.initialState;
                child2.stateDepth = parent.stateDepth+1;
                child2.parent = parent;
                child2.g = parent.g + 1;
                child2.direction = directions[1];
                child2.moveRight();
                child2.f = manhattanDistance(child2.getCurrentState()) +
child2.g;
                childStates.add(child2);
            }
            if (parent.isMoveDown()){
                BlockPuzzle child4 = new BlockPuzzle();
                //System.out.println("Down");
                child4.smileyPosition = parent.getSmileyPosition();
                child4.initialState = parent.getCurrentState();
                child4.currentState = child4.initialState;
                child4.stateDepth = parent.stateDepth+1;
                child4.parent = parent;
                child4.direction = directions[3];
                child4.moveDown();
                child4.g = parent.g + 1;
                //child4.moveRight();
                child4.f = manhattanDistance(child4.getCurrentState()) +
child4.g;
                childStates.add(child4);
            }
        }
        treeDepth = parent.stateDepth+1;
        return childStates;
    }

    /**
     * Simple method that calculates the Manhattan distance for each state
     * @param state of the puzzle
     * @return Integer that represents the Manhattan distance
     */
    private static Integer manhattanDistance(String state){
        int distance = 0;

        for (int i = 0; i < state.length(); i++) {
            if (state.charAt(i) == 'A'){
                distance+= Math.abs(i-5);
```

```
            }
            if (state.charAt(i) == 'B'){
                distance+=Math.abs(i-9);
            }
            if (state.charAt(i) == 'C'){
                distance+=Math.abs(i-13);
            }
        }

        return distance;
    }

    /**
     *
     * Simple boolean methods that check if movement is possible
     * @return true if possible, false if not possible
     */
    private boolean isMoveLeft(){

        return ((smileyPosition > 0 && smileyPosition <= 3) ||
(smileyPosition > 4 && smileyPosition <= 7)
                || (smileyPosition > 8 && smileyPosition <= 11) ||
(smileyPosition > 12 && smileyPosition <=15));


        //return smileyPosition > 0;
    }

    private boolean isMoveRight(){
        //smileyPosition < 15

         return ((smileyPosition >= 0 && smileyPosition < 3) ||
(smileyPosition >= 4 && smileyPosition < 7)
                || (smileyPosition >= 8 && smileyPosition < 11) ||
(smileyPosition >= 12 && smileyPosition < 15));



        //return smileyPosition < size-1;
    }

    private boolean isMoveUp(){
        //(smileyPosition <= 15 && smileyPosition >= 4)
        return smileyPosition <= size-1 && smileyPosition >=
Math.sqrt(size);
    }

    private boolean isMoveDown(){

        //(smileyPosition <= 11 && smileyPosition >= 0)
        return smileyPosition <= (size-1)-Math.sqrt(size) && smileyPosition
>= 0;
    }

    private boolean isMovementPossible(){
        return isMoveDown() || isMoveLeft() || isMoveRight() || isMoveUp();
    }

    /**
     * Used to compare the Manhattan distances of the states for the order
in the priority queue
```

```
     * @param o the second object that we compare
     * @return 1 if this.Object f > o.f, -1 if this.Object f < o.f, 0 if
equal
     */
    @Override
    public int compareTo(BlockPuzzle o) {
        if (this.f > o.f){
            return 1;
        } else if (this.f < o.f){
            return -1;
        } else {
            return 0;
        }
    }
}
```