1. Approach

The approach I took to solve the Blocksworld tile puzzle involves several steps.
Firstly, I have created a Node class, which is going to represent the state of the grid at any given point (i.e. the positions of the blocks/the grid's configuration). A state of the grid is represented by 3 properties – its configuration, a list of child states (the alternated states after moving left, right, up or down) and the state's parent (this is only needed for tracing the path to a solution). I have decided to implement the configuration as a 1D array as I think it is neater and slightly more efficient than a 2D array. The Node class implements Comparable because for A* heuristic search I am using a PriorityQueue, which compares nodes by their (cost so far + estimated cost to goal).

Secondly, I have created the Searcher class where I have implemented and tested the 4 search methods.

Depth-First Search

The original search method has obvious problems – it gets stuck in a loop. In order to avoid this I randomize the selected actions by shuffling the children of the expanded node. The result is that I get a solution very fast and with linear space complexity, however, it's not the most optimal (shortest distance from the root). I have implemented it by using a Stack.

Breadth-First Search

This search method guarantees to find an optimal solution given enough memory and time. The problem with it is that it keeps every node in memory so it becomes very slow with large number of nodes. It also visits already visited nodes, which adds a lot of unneeded nodes in memory and thus makes the time complexity a lot worse. I have implemented it by using a Queue.

Iterative Deepening Search

For implementing this search method I am using depth-limited search and on every iteration I increase the maximum depth by 1. Depth-limited search is similar to DFS so I have used a Stack

again and a limit for the depth. In order to stop expanding nodes when the depth limit is reached I check if the current node's depth equals the depth limit – if it doesn't then we know it would be less so we expand it, otherwise we don't expand it.

A* heuristic search

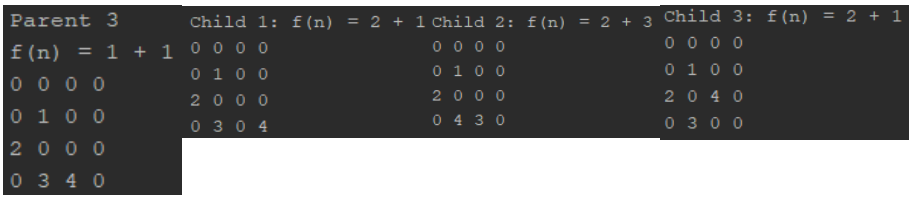
For implementing this search method I am using the Manhattan distance as a heuristic. When choosing what node to expand next I pick the one that has a minimum (cost so far + estimated Manhattan distance). The cost so far is determined by the number of moves that had to be taken in order to reach the current node, i.e. the depth of that node, and for the Manhattan distance I get the sum of the moves that need to be taken for each block to reach its final position. I have chosen to use a PriorityQueue to store the nodes in the fringe as it is the most efficient data structure for this method.

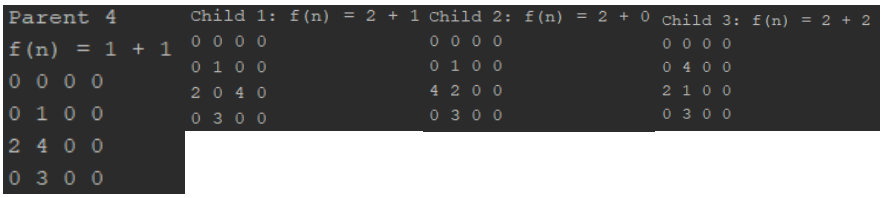
2. Evidence

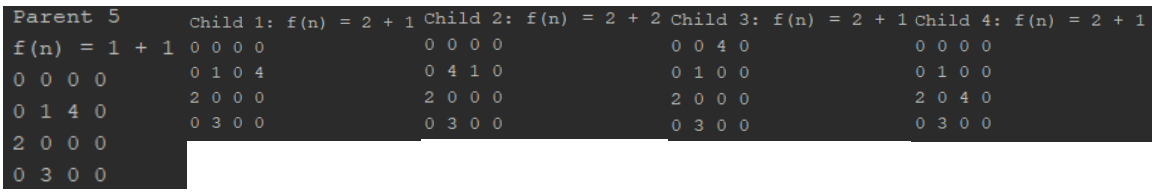
A* heuristic search

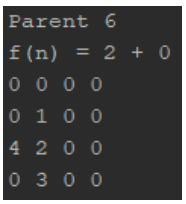
```
Parent 1        Child 1: f(n) = 1 + 1  Child 2: f(n) = 1 + 1  Child 3: f(n) = 1 + 1  Child 4: f(n) = 1 + 1
f(n) = 0 + 1    0 0 0 0                0 0 0 0                0 0 0 0                0 0 0 0
0 0 0 0         0 1 0 0                0 1 0 0                0 1 4 0                0 1 0 0
0 1 0 0         2 0 0 4                2 4 0 0                2 0 0 0                2 0 0 0
2 0 4 0         0 3 0 0                0 3 0 0                0 3 0 0                0 3 4 0
0 3 0 0
```

```
Parent 2        Child 1: f(n) = 2 + 1  Child 2: f(n) = 2 + 1  Child 3: f(n) = 2 + 1
f(n) = 1 + 1    0 0 0 0                0 0 0 0                0 0 0 0
0 0 0 0         0 1 0 0                0 1 0 4                0 1 0 0
0 1 0 0         2 0 4 0                2 0 0 0                2 0 0 0
2 0 0 4         0 3 0 0                0 3 0 0                0 3 0 4
0 3 0 0
```



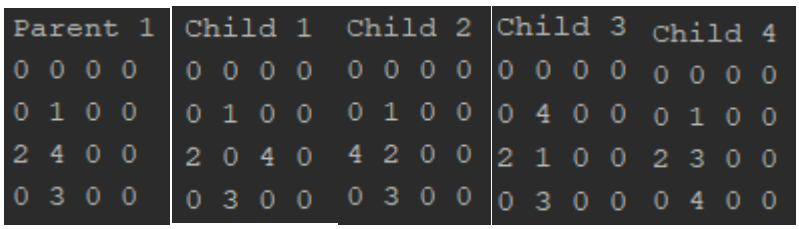


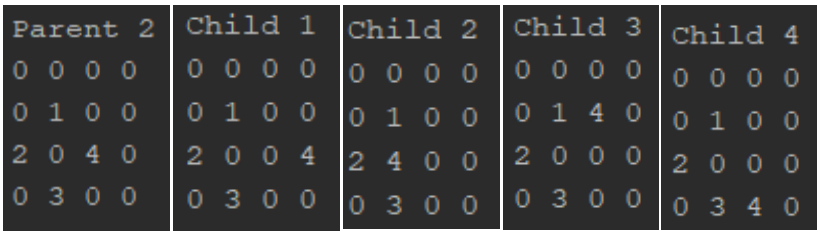


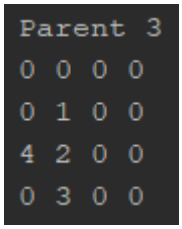


It can be observed that my initial state (Parent 1) has 4 children all with cost 2. Therefore, the next expanded node (Parent 2) will be any of them since their costs are equal. Parent 2 doesn't have a child with costs <= 2 thus the next node will be another child of Parent 1 (with cost 2). It turns out that only one of Parent 1's children has a child with cost <= 2, namely Parent 5. Therefore, that will be picked next after all of Parent 1's children and it is exactly the goal state so when we visit it the program halts as it is the goal state and no other node has a cost less than it. The images above prove that the A* method always takes the child in the PriorityQueue that has minimum cost. In order to show how it works, I have moved the start state closer to the goal state.

Breadth-First Search

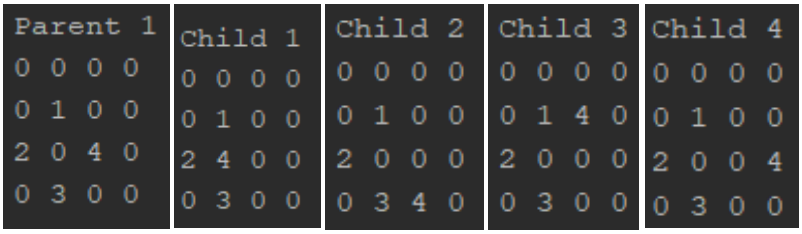




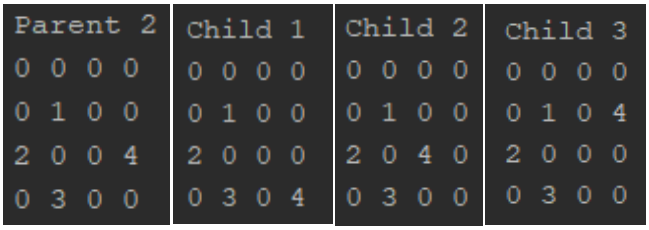


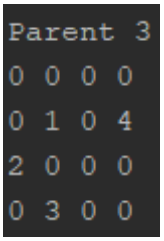
From the pictures above it can be observed that when choosing what node to expand next we always take the first one that we have seen, i.e. the first one from the queue. In this case I have moved the start state closer to the goal state again and it can be seen that Parent 2 is Parent 1's first child, then Parent 3 is Parent 1's second child and so on.

Depth-First Search

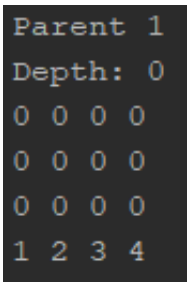


From the pictures it can be observed that when choosing what node to expand next we always take the last seen node. In this case Parent 2 is the last child of Parent 1, Parent 3 is the last child of Parent 2 and so on.

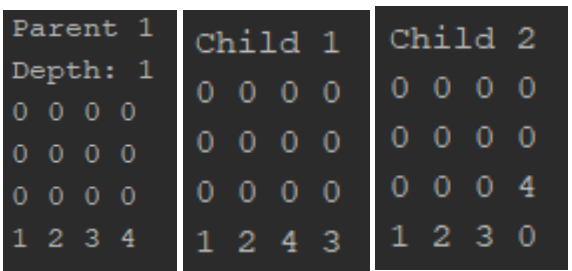




Iterative Deepening Search



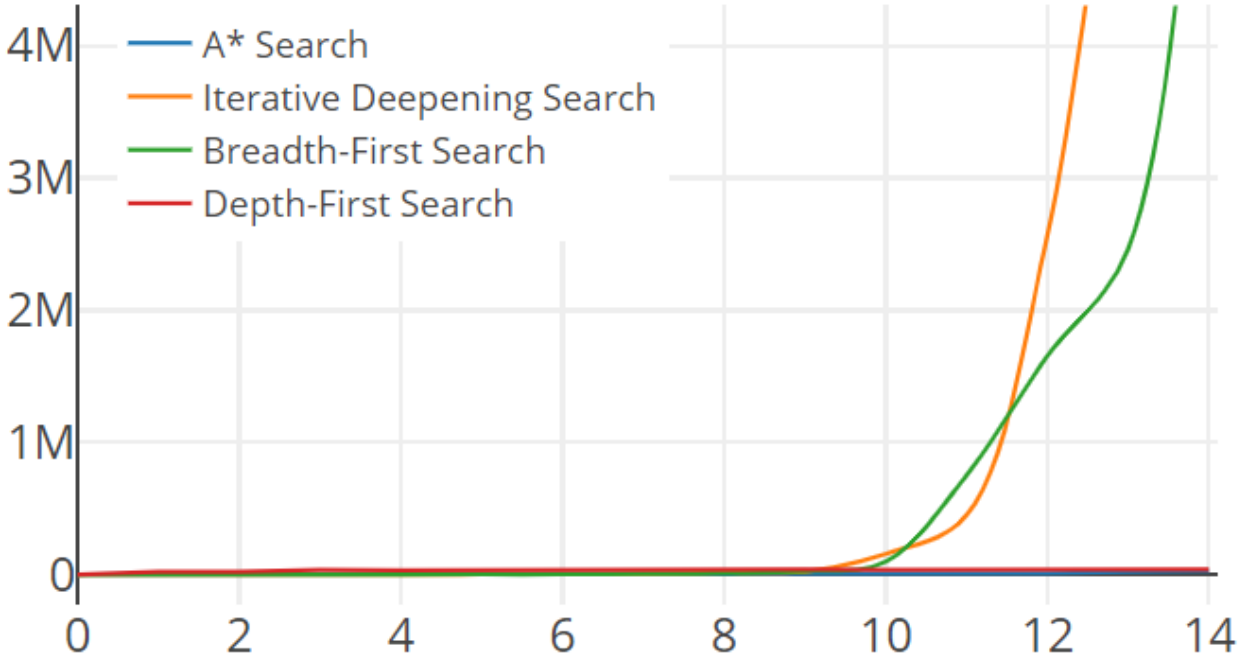
From the pictures it can be seen that on the first iteration the Parent node has no children as the depth is only 0. Therefore, then we visit only the parent node. On the next iteration, the depth becomes 1 so after the parent node we first visit the last seen child, namely Child 2 and then Child 1. Again we do not expand their children as the current depth is 1. With each increase of depth we do a depth limited search, which behaves like depth first search but has a depth limit. Therefore, on the next iteration we will visit the same nodes and also expand the children of Parent 2 and Parent 3.



```
Parent 2
Depth: 1
0 0 0 0
0 0 0 0
0 0 0 4
1 2 3 0
```

```
Parent 3
Depth: 1
0 0 0 0
0 0 0 0
0 0 0 0
1 2 4 3
```

3.Scalability study



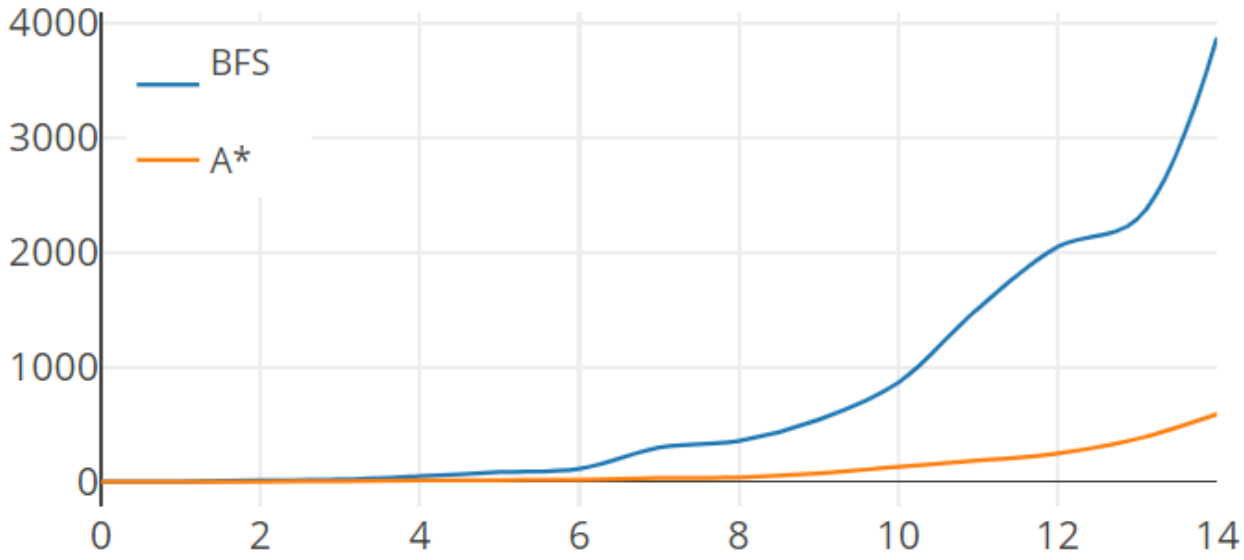
The approach I took to control the problem difficulty was by controlling the depth of the solution, i.e. by moving the start state closer to the goal state and observing the time complexity (how many nodes will have to be expanded to reach the final state) of each search method from there. On the plot I compare the time complexity of the search methods where on the x-axis I have the problem difficulty (the depth of the solution) and on the y-axis is the number of nodes expanded to find the solution. Note that Depth-First Search on the plot uses randomizing of the actions taken and I have estimated the average number of nodes expanded for each problem difficulty by running the method 100 times on each level and taking the mean of the results. It can be seen that with scaling of the problem difficulty BFS's and IDS's performance significantly drops meaning that the nodes they have to visit becomes very large. Even though IDS has to expand more nodes than BFS it compensates by having linear space complexity as it only has to keep **O(bd)** nodes (where b is the branching factor and d is the current depth) in memory whereas BFS has to keep all nodes in memory. Furthermore, it can be seen that A* is the most efficient search method when using a good heuristic (in my case I am using the cost so far + Manhattan distance) as it finds an optimal solution and expands significantly less nodes than IDS or BFS. When comparing it to DFS we see that, even though DFS has slightly more nodes expanded, it doesn't guarantee to find an optimal solution. Moreover, DFS's actions are randomized else it would be unable to solve the problem as it would get stuck in infinite loops. A plus that DFS has is that it has a linear space complexity, whereas A* has to keep all nodes in

memory. When comparing A* to IDS we see that the latter expands a lot more nodes, however, for sufficiently large problems, even though slowly, IDS will find an optimal solution whereas A* can run out of memory.

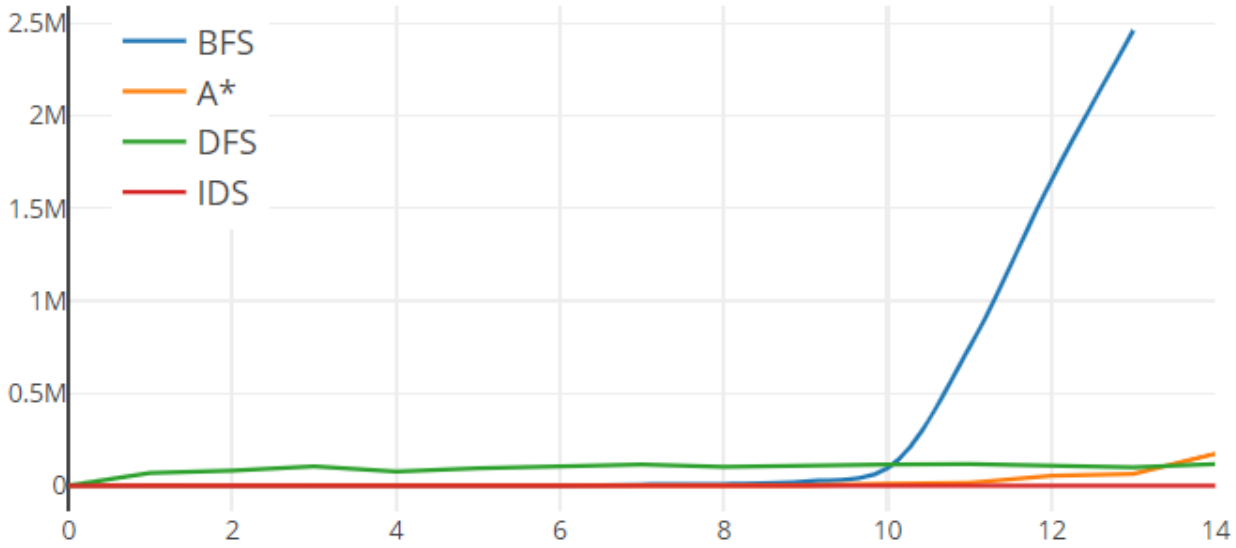
4. Extras and limitations

One additional think that I have done in order to improve the efficiency of the methods is to implement graph search. Graph search is different from the tree search that I have discussed so far because it keeps track of already visited nodes. In this way we avoid unnecessary computation. I think this is a sensible thing to do as it makes the expanded nodes significantly less. The implementation of the graph searches is left commented in my code.

From this plot we can see how much keeping track of visited nodes lowers the number of expanded nodes in BFS and A*, which both have to store all visited nodes in memory.



Another additional thing that I have done and think is useful to see is to plot the space complexity of each search method. This can be seen in the figure below.



Comparing all of the search methods' space complexities it can be observed that BFS has the worst space complexity as it has to keep every node in the memory. A* also has to keep every expanded node in memory, however, the number of those nodes is significantly less if we have a good heuristic. In terms of DFS and IDS, they both have O(bd) space complexity, however, in our case DFS stores significantly more nodes since the depth of the solution is often quite big. In order to measure the space complexity of DFS I have ran it 100 times for each problem difficulty and have taken the mean of the results. IDS has the best space complexity out of all of the 4 search methods.

5. References

6. Code

Node class

```java
import java.util.ArrayList;
import java.util.List;

public class Node implements Comparable<Node> {
    private int[] configuration;
    private List<Node> children = new ArrayList<>();
    private Node parent;
    private int agentPosition;

    public Node(int[] configuration) {
        this.configuration = new int[configuration.length];
        copy(configuration,this.configuration);
    }

    public void expandAll(){

        // find agent position
        for(int i = 0; i < configuration.length; i++){
            if(configuration[i] == 4){
                agentPosition = i;
            }
        }

        moveRight(configuration,agentPosition);
        moveLeft(configuration,agentPosition);
        moveUp(configuration,agentPosition);
        moveDown(configuration,agentPosition);
    }

    public void moveLeft(int[] configuration, int pos){
        if(pos % 4 != 0){
            int[] childConfiguration = new int[configuration.length];
            copy(configuration,childConfiguration);

            int temp = childConfiguration[pos];
            childConfiguration[pos] = childConfiguration[pos - 1];
            childConfiguration[pos - 1] = temp;

            Node child = new Node(childConfiguration);
            this.children.add(child);
            child.setParent(this);
        }
    }

    public void moveRight(int[] configuration, int pos){
        if(pos % 4 != 3){
            int[] childConfiguration = new int[configuration.length];
            copy(configuration,childConfiguration);

            int temp = childConfiguration[pos];
            childConfiguration[pos] = childConfiguration[pos + 1];
            childConfiguration[pos + 1] = temp;

            Node child = new Node(childConfiguration);
            this.children.add(child);
            child.setParent(this);
        }
    }
```

```
    public void moveUp(int[] configuration, int pos){
        if(pos > 3){
            int[] childConfiguration = new int[configuration.length];
            copy(configuration,childConfiguration);

            int temp = childConfiguration[pos];
            childConfiguration[pos] = childConfiguration[pos - 4];
            childConfiguration[pos - 4] = temp;

            Node child = new Node(childConfiguration);
            this.children.add(child);
            child.setParent(this);
        }
    }

    public void moveDown(int[] configuration, int pos){
        if(pos < 12){
            int[] childConfiguration = new int[configuration.length];
            copy(configuration,childConfiguration);

            int temp = childConfiguration[pos];
            childConfiguration[pos] = childConfiguration[pos + 4];
            childConfiguration[pos + 4] = temp;

            Node child = new Node(childConfiguration);
            this.children.add(child);
            child.setParent(this);
        }
    }

    private void copy(int[] original, int[] copy){
        System.arraycopy(original, 0, copy, 0, original.length);
    }

    // calculates how many moves are needed for a block to reach its final position
    public int getDistanceToFinal(int num, int finalPosition){
        int index = findElement(this.getConfiguration(),num);

        if(index % 4 == finalPosition % 4){
            return Math.abs((index - finalPosition)/4);
        }else if(index % 4 == finalPosition % 3){
            return 1 + Math.abs((index + 1 - finalPosition)/4);
        }else if(index % 4 == finalPosition % 5){
            return 1 + Math.abs((index - 1 - finalPosition)/4);
        }else {
            return 2 + Math.abs((index - 2 - finalPosition)/4);
        }
    }

    // calculate the sum of the moves needed for A,b and C to their final positions
    public int getSumOfDistances(){
        int sum = 0;
        int[] elements = new int[3];
        elements[0] = 1;
        elements[1] = 2;
        elements[2] = 3;

        int[] finalPositions = new int[3];
        finalPositions[0] = 5;
        finalPositions[1] = 9;
        finalPositions[2] = 13;

        for(int i = 0; i < elements.length; i++){
```

```
            sum += getDistanceToFinal(elements[i], finalPositions[i]);
        }
        return sum;
    }

    // compare by sum of distances
    @Override
    public int compareTo(Node o) {
        if(this.getSumOfDistances() + this.tracePath().size()-1 >
o.getSumOfDistances() + o.tracePath().size()-1){
            return 1;
        }else if(this.getSumOfDistances() + this.tracePath().size()-1 <
o.getSumOfDistances() + o.tracePath().size()-1){
            return -1;
        }
        return 0;
    }

    // trace how we reached a given node
    public List<Node> tracePath(){
        List<Node> path = new ArrayList<>();
        Node currentNode = this;
        path.add(currentNode);

        while(currentNode.getParent() != null){
            path.add(currentNode.getParent());
            currentNode = currentNode.getParent();
        }
        return path;
    }

    public int findElement(int[] configuration, int element){
        for(int i = 0; i < configuration.length; i++){
            if(configuration[i] == element){
                return i;
            }
        }
        return -1;
    }

    public boolean isGoal(){
        return this.configuration[5] == 1 && this.configuration[9] == 2 &&
this.configuration[13] == 3;
    }

    public boolean isSameConfiguration(int[] configuration){
        boolean isSame = true;

        for(int i = 0; i < configuration.length; i++){
            if(this.configuration[i] != configuration[i]){
                isSame = false;
            }
        }
        return isSame;
    }

    public void printConfiguration(){
        for(int i = 0; i < this.configuration.length; i++){

            if((i + 1) % 4 == 0){
                System.out.print(this.configuration[i]);
                System.out.println();
            }else{
```

```
                System.out.print(this.configuration[i] + " ");
            }
        }
    }

    private void setParent(Node parent) {
        this.parent = parent;
    }

    public List<Node> getChildren() {
        return children;
    }

    public void resetChildren(){
        this.children.clear();
    }

    public int[] getConfiguration() {
        return configuration;
    }

    public Node getParent() {
        return parent;
    }
}
```

Searcher class

```
import java.util.*;

public class Searcher {
    private int[] startState = new int[16];
    private int count = 1;
    private List<Integer> avg = new ArrayList<>(); //stores dfs results

    public static void main(String[] args) {
        Searcher s = new Searcher();
        s.init();
        Node root = new Node(s.startState);
        long startTime = System.nanoTime();
        List<Node> solutions = s.aStar(root);
        s.printPath(solutions);
        s.dfsAverage(root);
        long endTime = System.nanoTime();
        System.out.println((double)(endTime - startTime)/1000000000);
    }

    private void dfsAverage(Node root){
        List<Node> solutions = this.aStar(root);
        this.printPath(solutions);
        for(Node n : solutions){
            //run dfs 100 times to get an average
            for(int i = 0; i < 100; i++){
                Node newNode = new Node(n.getConfiguration());
                this.dfs(newNode);
```

```
            }
        }
    }

    // this method shows how I count expanded nodes in IDS
    private void idsCount(Node root){
        List<Node> solutions = this.aStar(root);
        this.printPath(solutions);
        for(Node n : solutions){
             Node newNode = new Node(n.getConfiguration());
             this.count = 0;
             this.ids(newNode);
        }
    }

    private void init(){
        for(int i = 0; i < this.startState.length; i++){
            if(i == 12){
                this.startState[12] = 1;
            }else if(i == 13){
                this.startState[13] = 2;
            }else if(i == 14){
                this.startState[14] = 3;
            }else if(i == 15){
                this.startState[15] = 4;
            }else{
                this.startState[i] = 0;
            }
        }
    }

    private void testInit(){
        for(int i = 0; i < startState.length; i++){
            if(i == 5){
                startState[i] = 1;
            }else if(i == 8){
                startState[i] = 2;
            }else if(i == 13){
                startState[i] = 3;
            }else if(i == 11){
                startState[i] = 4;
            }else{
                startState[i] = 0;
            }
        }
    }

    private List<Node> ids(Node root){
        for(int depth = 0; depth < Integer.MAX_VALUE; depth++){
            Node found = dls(root,depth);
            if(found != null){
                return found.tracePath();
            }
        }
        return null;
    }

    private Node dls(Node root, int depth){
        root.resetChildren();
        //visited list is only used for graph search
       // List<Node> visited = new ArrayList<>();
        Stack<Node> toVisit = new Stack<>();
        toVisit.push(root);
```

```
        while(!toVisit.isEmpty()){
            Node current = toVisit.pop();
            this.count++;
           // visited.add(current);

            if(current.isGoal()){
                System.out.println("Goal found");
                System.out.println(this.count);
                return current;
            }

            // check if max depth is reached
            if(current.tracePath().size() - 1 != depth) {
                current.expandAll();

                for (Node child : current.getChildren()) {
                    // this check is only done for graph searches
//                      if(!hasNode(visited,child) && !hasNode(toVisit,child)){
//                          toVisit.push(child);
//                      }
                    toVisit.push(child);
                }
            }
        }

        return null;
    }

    private  List<Node> dfs(Node root){
        //visited list is only used for graph search
       // List<Node> visited = new ArrayList<>();
        Stack<Node> toVisit = new Stack<>();
        List<Node> path = new ArrayList<>();
        boolean goalFound = false;

        toVisit.push(root);
        int count = 0;

        while (!goalFound){
            Node current = toVisit.pop();
          // visited.add(current);
            count++; // counts the number of expanded nodes

            if(current.isGoal()){
                System.out.println("Goal found");
                goalFound = true;
                path = current.tracePath();
                avg.add(count);
            }else{
                current.expandAll();
                Collections.shuffle(current.getChildren());

                for(Node child : current.getChildren()){
                    // this check is only done for graph searches
//                      if(!hasNode(visited,child) && !hasNode(toVisit,child)){
//                          toVisit.push(child);
//                      }
                    count++;
                    toVisit.push(child);
                }
            }
        }
```

```
        // get the average of 100 results
        if(avg.size() == 100){
            System.out.println(Math.round(avg.stream().mapToInt(val ->
val).average().orElse(0.0)));
            avg.clear();
        }

        return path;
    }

    private List<Node> aStar(Node root){
        //visited list is only used for graph search
       // List<Node> visited = new ArrayList<>();
        PriorityQueue<Node> toVisit = new PriorityQueue<>();
        List<Node> path = new ArrayList<>();
        boolean goalFound = false;

        toVisit.add(root);
        int count = 0;

        while(!goalFound){
            Node currentNode = toVisit.poll();
            count++;
           // visited.add(currentNode);

            currentNode.expandAll();

            if(currentNode.isGoal()) {
                System.out.println("Goal found");
                goalFound = true;
                path = currentNode.tracePath();
                System.out.println(count);
            }else{
                for(Node child : currentNode.getChildren()){
                    // this check is only done for graph searches
//                      if(!hasNode(visited,child) && !toVisit.contains(child)){
//                          toVisit.add(child);
//                      }

                    toVisit.add(child);
                }
            }
        }

        return path;
    }

    private List<Node> bfs(Node root){
        Queue<Node> toVisit = new LinkedList<>();
        //visited list is only used for graph search
       // List<Node> visited = new ArrayList<>();
        List<Node> path = new ArrayList<>();
        boolean goalFound = false;

        toVisit.add(root);
        int count = 0;

        while(!goalFound){
            Node currentNode = toVisit.poll();
            count++;
           // visited.add(currentNode);
```

```
            if(currentNode.isGoal()) {
                System.out.println("Goal found");
                goalFound = true;
                path = currentNode.tracePath();
                System.out.println(count);
            }
            currentNode.expandAll();

            for(Node child : currentNode.getChildren()){
                // this check is only done for graph searches
//                  if(!hasNode(visited,child) && !toVisit.contains(child)){
//                      toVisit.add(child);
//                  }

                toVisit.add(child);
            }
        }

        return path;
    }

    private boolean hasNode(List<Node> list, Node n){
        boolean hasNode = false;

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

            if(list.get(i).isSameConfiguration(n.getConfiguration())){
                hasNode = true;
            }
        }

        return hasNode;
    }

    private void printPath(List<Node> path){
        for(Node n : path){
            n.printConfiguration();
            System.out.println();
        }
    }
}
```