Comp2208 Assignment: Search Methods

Investigation of scalability with problem difficulty of

depth first, breadth first, iterative deepening, and A* heuristic search

Valeriu Andrei Cristea

ID: 28335988 / vac1u16

Contents

1. Approach

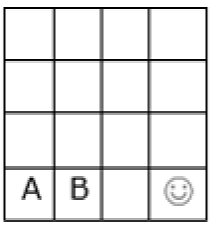
I chose Java for this assignment as the most intuitive approach involves object oriented programming and Java is the object-oriented language I have the most experience with.

I consider the starting state to be A in the bottom left corner and following letter are placed on the right of the previous letter until the desired number of blocks is reached. The agent starts from the bottom right corner. The number of blocks is always smaller than the size of the grid.

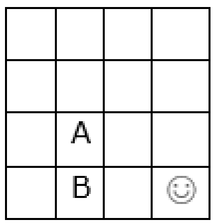
I consider the goal state to be all blocks on the second column, ordered alphabetically, if read top to bottom, with the last letter sitting on the bottom row; agent can be anywhere.

I consider a search to fail if it takes longer than 15 minutes or it runs out of memory.

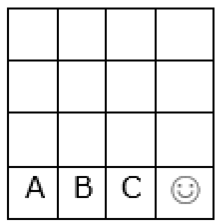
Examples:



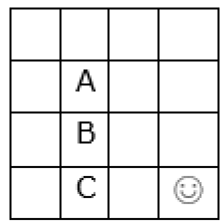
Start:
size=4
numberOfBlocks=2



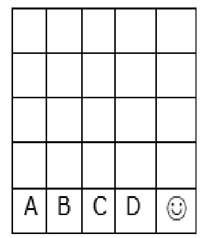
Goal:
size=4
numberOfBlocks=2



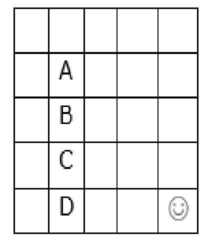
Start:
size=4
numberOfBlocks=3



Goal:
size=4
numberOfBlocks=3



Start:
size=5
numberOfBlocks=4



Goal:
size=5
numberOfBlocks=4

Implementation:

Description of classes and their methods:

1. Block:

- Represents the blocks (A, B, etc.) and the agent
- Stores their position and their name (agent or A, B, etc.)

2. Node:

- Represents a node for the tree search
- Stores the blocks in an ArrayList for scalability, the move used to get to it from the parent node, its parent node, the size of the grid and how many blocks there are

3. Main:

- Contains the search methods and methods to write the outputs they produce to a file
- the `main(String args[])` method acts as a test harness for the methods

Methods:

- Constructor for when the node is the root and when it is a child
- `void makeMove(char move)` , used in the child constructor, moves the agent if possible and swaps positions with a block if they occupy the same position
- `boolean isSolution()` , checks if the node is a solution, general implementation for scalability
- `ArrayList<Node> getMoves()` , returns the nodes created after taking each possible move

2. Evidence of the 4 search methods in operation

2.1 BFS

Uses a Queue to store the nodes that are going to be checked. It starts by offering the starting node to the Queue. While the Queue is not empty and a solution has not been found it polls a node from the Queue and checks if it's a solution. If it is it prints the time and nodes expanded and stops the BFS, otherwise offers all the nodes created by using `getMoves()` on the current node to the Queue.

Could be improved by making it a graph search by adding a HashSet, adding all the nodes explored to the HashSet, and checking if a node already belongs to the HashSet before offering it to the Queue.

2.2 DFS

Similar implementation to the BFS except it uses a Stack instead of the Queue and shuffles the ArrayList of nodes returned by `getMoves()` before pushing the nodes from the ArrayList to the Stack.

Since it is not a graph search it does not check if a node has already been explored so the Stack is unnecessary as the next node it checks will always be the first element of the shuffled ArrayList.

Could be improved the same way as the BFS by making it a graph search, then the Stack would be needed as well.

2.3 Iterative deepening search

Algorithm adopted form pseudocode (En.wikipedia.org, 2017). It starts with a maximum depth of 0 and the starting Node. It calls the deepening function on the starting Node and it increases the maximum depth until a solution is found.

The deepening function checks if it is allowed to go deeper. If it is not allowed it checks if the solution is the current node and if it's not it returns null. If it is allowed to go deeper the function is recursively called on the nodes created by the `getMoves()` method and the allowed depth is decremented by one. If the deepening function called on these moves is not null it means that a solution has been found.

It is a modified version of DFS algorithm where the maximum depth increases until a solution is found.

2.4 A*

A* algorithm is a modified version of Dijkstra's algorithm, also adapted for a tree in this situation, that uses heuristics. It considers nodes with a lower cost to be more promising. The cost of a node is sum of the Manhattan Distance from each block to its corresponding position in the solution and the depth of the node.

It functions the same way as BFS except it uses a PriorityQueue instead of a Queue.

Could be improved by making it a graph search the same way as BFS.

Additional note about all methods: they could be drastically improved if `getMoves()` doesn't return the opposite of the last move made (A* being able to solve with size 11 and 3 blocks under 10 seconds ; it takes A* over 15 minutes to solve for size 9 and 3 blocks without this), but the current implementation is the simplest. I expand on this in Extras 4.1.1.

2.5 Outputs (U = up; D = down; L = left; R = right)

Original problem output:

```
Size is 4X4
Number of Blocks is 3

BFS time = 49338 ms
Nodes expanded : 5423596
Solution path is : U L L D L U R D R U U L D L

DFS time = 29 ms
Nodes expanded : 10339
Solution path is : L U L R D R U L L R L R U R U D U

IDS time = 2898 ms
Nodes expanded : 7849127
Solution path is : U L L D L U R D R U U L D L

A* time = 41 ms
Nodes expanded : 20421
Solution path is : U L L D L U R D R U U L D L
```

DFS output not shown as it is 10339 moves long.

More output example (DFS solution not shown):

```
Size is 3X3
Number of Blocks is 3

BFS time = 668 ms
Nodes expanded : 259297
Solution path is : L L U R D R U U L D D R

DFS time = 0 ms
Nodes expanded : 1664

IDS time = 69 ms
Nodes expanded : 398687
Solution path is : L L U R D R U U L D D R

A* time = 16 ms
Nodes expanded : 1750
Solution path is : L L U R D R U U L D D R
```

```
Size is 5X5
Number of Blocks is 3
BFS failed because  15 minute exceeded

DFS time = 339 ms
Nodes expanded : 230928

IDS time = 4599 ms
Nodes expanded : 47447008
Solution path is : U L L L D L U R D R U U L D L

A* time = 37 ms
Nodes expanded : 63625
Solution path is : U L L L D L U R D R U U L D L
```

```
Size is 8X8
Number of Blocks is 3

DFS time = 15373 ms
Nodes expanded : 6081356

IDS time = 325845 ms
Nodes expanded : 3901577720
Solution path is : U L L L L L L D L U R D R U U L D L

A* time = 6842 ms
Nodes expanded : 4187489
Solution path is : L U L L L L L D L U R D R U U L D L
```

```
Size is 5X5
Number of Blocks is 4

DFS time = 14439 ms
Nodes expanded : 5092132
IDS failed because  15 minute exceeded
A* failed because  15 minute exceeded

Size is 6X6
Number of Blocks is 4

DFS time = 16698 ms
Nodes expanded : 8839779

Size is 7X7
Number of Blocks is 4
DFS failed because  15 minute exceeded
```

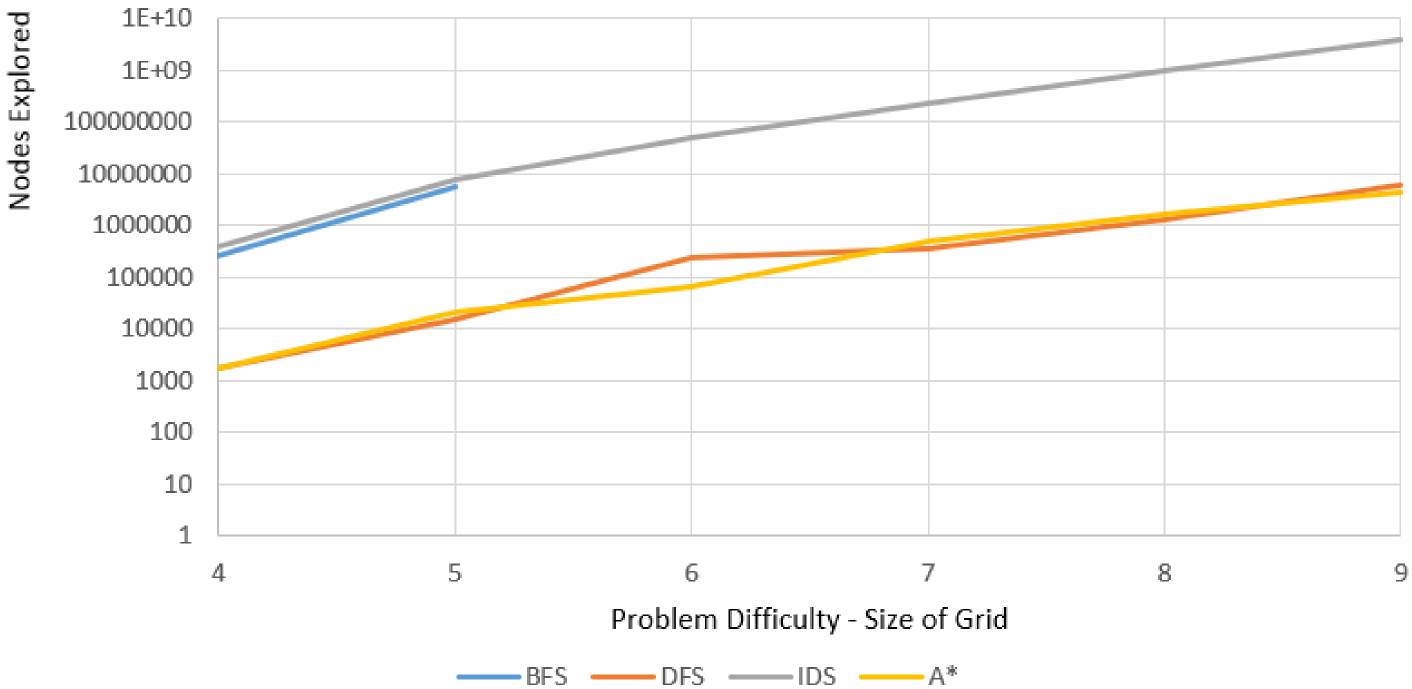
Full output of size 4 and 3 blocks at the end.

3. Scalability Study

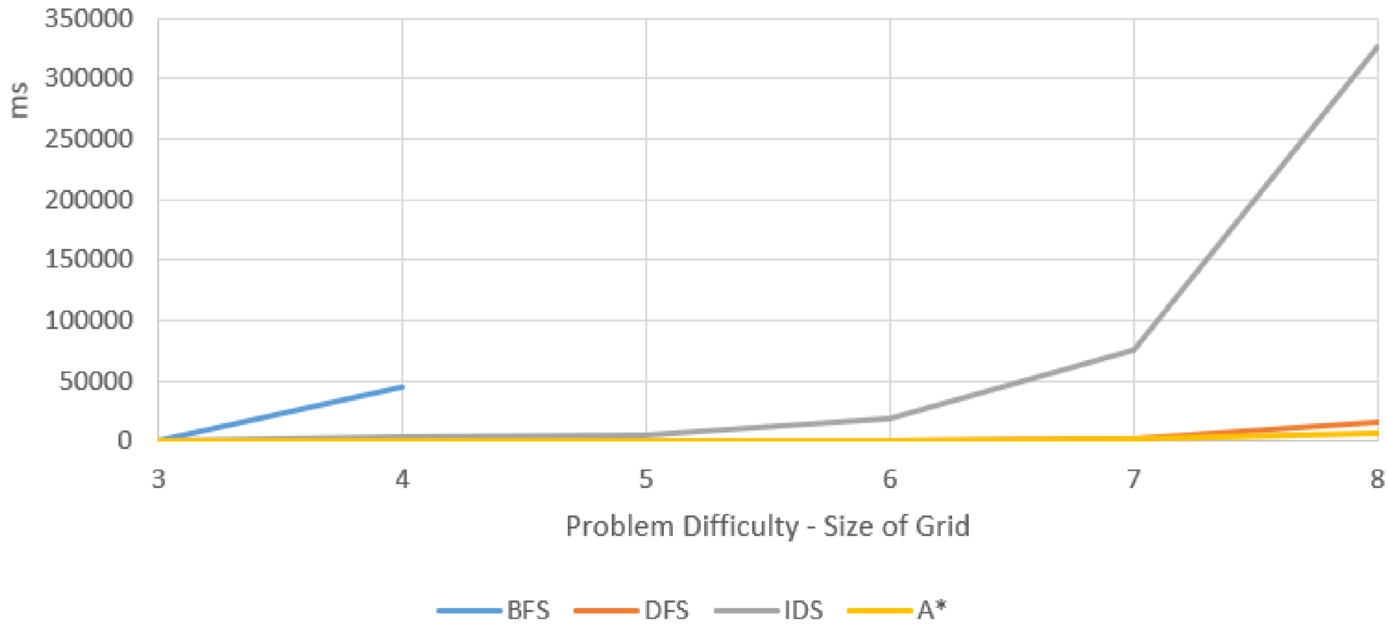
3.1 Description

My implementation general and works for any size > 2 and numberOfBlocks < size.
I do not have a test harness class as I test all my methods in Main and they write the output to a file which I save.
I control the problem difficulty by altering the size of the grid in the graphs below.

Graph 1: Problem Difficulty (size of grid) vs Time Complexity (nodes explored) ($\log_{10}$ scale)



Graph 2: Problem Difficulty (size of grid) vs Time Complexity (time taken; in ms)



From this data and these graphs, I can interpret and conclude the following:

- BFS: It's the slowest and worst algorithm when the size is larger than 5, as it does not even finish the search in under 15 minutes. When the size is smaller than 5 it performs slightly better than IDS and worse than A* and DFS. It always finds the optimal solution compared to DFS.

- DFS: It's a better algorithm than BFS and IDS on average but the downside is that it's completely random. It usually fails around size 7 – 9 but because of it's random nature it is possible to fail at sizes smaller than 7 or larger than 8. The solution it finds will never be the optimal compared to the other 3 algorithms.

- Iterative Deepening Search: It performs better than BFS and worse than A* and DFS. It suffers from the same problem as BFS, namely it explores too many nodes. As the first graph shows, IDS and BFS explore as many nodes when the size is 4 as does DFS and A* when the size is 7. The solution it finds is always optimal.

- A*: Best algorithm in terms of nodes explored and time taken. Always gives the best solution and is more reliable compared to DFS.

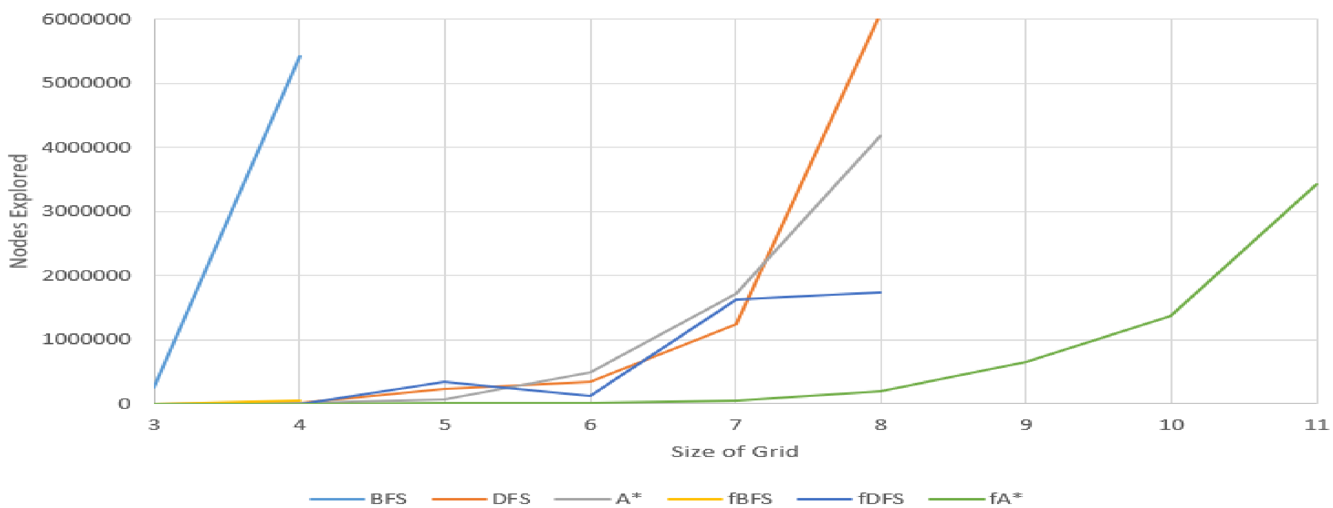
4. Extras and Limitations

4.1 Extras

4.1.1 `getMoves()` improvement: `getFastMoves()` - returns the nodes created after taking each possible move without the node created by going back to parent state.

4.1.2 Improving the efficiency by making the tree searches (BFS, DFS and A*) into graph searches by using HashSet and overriding the hashCode method in Block and Node and using `getFastMoves()` instead `getMoves()`.(methods are called fbfs, fdfs, fastar-fastar has no HashSet)

Note: A* is faster if I use `getFastMoves()` and no HashSet. With or without HashSet it explores the same number of nodes but substantially faster without – over 5 minutes for 7x7 with HashSet and under 1 minute for 11x11 without.



Note on the graph: I consider fDFS, fBFS and fA* to fail if they take more than 5 minutes, the others still more than 15 minutes.

Interpretation of the graph: As expected A* graph search is the optimal method. fBFS times out at size 5. fDFS has random results as it still shuffles the ArrayList. fDFS and A* both fail at size 9.

4.1.3 Adding immovable blocks:
- added `getMovesObstacles()` returns the ArrayList of nodes created after taking each possible move.
- added a new Node Constructor that `getMovesObstacles()` calls when it creates the nodes that it adds to the ArrayList
- methods are called ibfs, idfs, iastar and their implementation is the same as fbfs, fdfs, fastar except they call `getMovesObstacles()` instead of `getFastMoves()`.

4.2 Limitations and weakness of work

A better heuristic method to calculate the cost from the node to the solution could improve A*.

Writing IDS to not be recursive and use a Stack could help with time and would prevent possible stack overflow error.

Using a faster programming language could result in better performance space and time wise but the implementation might be more difficult to achieve.

When gathering the data for the report I could have tested DFS more times and made an average for better results.

I assume that the methods only receive valid input. I could check for valid input when calling the methods for making possible the adaptation of the classes in different programs.

The program is heavily limited in speed by the CPU and heap memory. An implementation that uses less data structures and less object-oriented programming could achieve better space efficiency and possibly better time as well.

My implementation could also be improved by adding a TestHarness class that tests the algorithms for various sizes and logs the errors and outputs, eliminating the need of having the main method cluttered with comments.

5. Reference

(En.wikipedia.org, 2017). https://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search

6. Code

Block Class

```
public class Block {

    //no getters and setters is a deliberate design choice
    public char name;
    public int x, y;

    public Block(char name, int x, int y) {
        this.name = name;
        this.x = x;
        this.y = y;
    }

    public int hashCode(){
        Character c = name;
        int sum = c.hashCode();
        sum = sum*17+ x;
        sum = sum*17+ y;
        return sum;
    }
}
```

Node Class

```java
import java.util.ArrayList;

/**
 * Created by Andr on 28-Nov-17.
 */
public class Node implements Comparable {

    Node parent;
    //blocks.get(0) is always the agent
    ArrayList<Block> blocks;
    //     y
    //    --->
    // x |
    //    v
    char lastMove;
    int size, noBlocks, cost, depth;


    //root
    public Node(int size, int noBlocks) {
        this.size = size;
        this.noBlocks = noBlocks;
        blocks = new ArrayList<Block>();
        blocks.add(new Block('1', size - 1, size - 1));
        char tmp;
        for (int i = 0; i < noBlocks; i++) {
            tmp = 'A';
            tmp += i;
            blocks.add(new Block(tmp, size - 1, i));
        }
    }

    //node
    public Node(Node parent, char move) {
        this.parent = parent;
        this.lastMove = move;
        this.size = parent.getSize();
        this.noBlocks = parent.getNoBlocks();
        blocks = new ArrayList<>();
        for (Block b : parent.getBlocks())
            blocks.add(new Block(b.name, b.x, b.y));
        this.makeMove(move);
    }

    public Node(Node parent, char move, boolean hasBlocks) {
        this.parent = parent;
        this.lastMove = move;
        this.size = parent.getSize();
        this.noBlocks = parent.getNoBlocks();
        blocks = new ArrayList<>();
        for (Block b : parent.getBlocks())
            blocks.add(new Block(b.name, b.x, b.y));
        this.makeMoveObstacle(move);
    }

    //used for test
    //creates the solution for that size
    public Node(int size, int noBlocks, boolean isSolution) {
        this.size = size;
        blocks = new ArrayList<Block>();
        blocks.add(new Block('1', size - 1, size - 1));
        this.noBlocks = noBlocks;
```

```java
        //agent's pos is irrelevant
        char tmp;
        for (int i = 0; i < noBlocks; i++) {
            tmp = 'A';
            tmp += i;
            blocks.add(new Block(tmp, i + size - noBlocks, 1));
        }
    }

    @SuppressWarnings("Duplicates")
    private void makeMove(char move) {

        switch (move) {
            case 'U': //up  x--
                blocks.get(0).x--;
                //checking if the agent is over a block
                //it can be over at most a block
                for (int i = 1; i < blocks.size(); i++)
                    if (blocks.get(i).y == blocks.get(0).y)
                        if (blocks.get(i).x == blocks.get(0).x) {  //they have to swap
                            blocks.get(i).x++;
                            return;
                        }
                break;

            case 'D': //down x++
                blocks.get(0).x++;
                //checking if the agent is over a block
                //it can be over at most a block
                for (int i = 1; i < blocks.size(); i++)
                    if (blocks.get(i).y == blocks.get(0).y)
                        if (blocks.get(i).x == blocks.get(0).x) {  //they have to swap
                            blocks.get(i).x--;
                            return;
                        }
                break;

            case 'L': //left y--
                blocks.get(0).y--;
                //checking if the agent is over a block
                //it can be over at most a block
                for (int i = 1; i < blocks.size(); i++)
                    if (blocks.get(i).y == blocks.get(0).y)
                        if (blocks.get(i).x == blocks.get(0).x) {  //they have to swap
                            blocks.get(i).y++;
                            return;
                        }
                break;

            case 'R': //right y++
                blocks.get(0).y++;
                //checking if the agent is over a block
                //it can be over at most a block
                for (int i = 1; i < blocks.size(); i++)
                    if (blocks.get(i).y == blocks.get(0).y)
                        if (blocks.get(i).x == blocks.get(0).x) {  //they have to swap
                            blocks.get(i).y--;
                            return;
                        }
                break;
        }
    }

    @SuppressWarnings("Duplicates")
    private void makeMoveObstacle(char move) {

        switch (move) {
            case 'U': //up  x--
                //checking if the agent is over a block
                //it can be over at most a block
```

```java
            blocks.get(0).x--;
            for (int i = 1; i <= noBlocks; i++)
                if (blocks.get(i).y == blocks.get(0).y)
                    if (blocks.get(i).x == blocks.get(0).x) {//they have to swap
                        blocks.get(0).x--;
                        blocks.get(i).x++;
                        return;
                    }

            //checks if there is an immovable block
            for (int i = noBlocks + 1; i < blocks.size(); i++)
                if (blocks.get(i).y == blocks.get(0).y)
                    if (blocks.get(i).x == blocks.get(0).x) {  //found block
                        blocks.get(0).x++;  //agent goes back
                        return;
                    }
            break;

        case 'D': //down x++
            blocks.get(0).x++;
            //checking if the agent is over a block
            //it can be over at most a block
            for (int i = 1; i <= noBlocks; i++)
                if (blocks.get(i).y == blocks.get(0).y)
                    if (blocks.get(i).x == blocks.get(0).x) {  //they have to swap
                        blocks.get(i).x--;
                        return;
                    }

            //checks if there is an immovable block
            for (int i = noBlocks + 1; i < blocks.size(); i++)
                if (blocks.get(i).y == blocks.get(0).y)
                    if (blocks.get(i).x == blocks.get(0).x) { //found block
                        blocks.get(0).x--;  //agent goes back
                        return;
                    }
            break;

        case 'L': //left y--
            blocks.get(0).y--;
            //checking if the agent is over a block
            //it can be over at most a block
            for (int i = 1; i <= noBlocks; i++)
                if (blocks.get(i).y == blocks.get(0).y)
                    if (blocks.get(i).x == blocks.get(0).x) {  //they have to swap
                        blocks.get(i).y++;
                        return;
                    }

            //checks if there is an immovable block
            for (int i = noBlocks + 1; i < blocks.size(); i++)
                if (blocks.get(i).y == blocks.get(0).y)
                    if (blocks.get(i).x == blocks.get(0).x) { //found block
                        blocks.get(0).y++;  //agent goes back
                        return;
                    }
            break;

        case 'R': //right y++
            blocks.get(0).y++;
            //checking if the agent is over a block
            //it can be over at most a block
            for (int i = 1; i <= noBlocks; i++)
                if (blocks.get(i).y == blocks.get(0).y)
                    if (blocks.get(i).x == blocks.get(0).x) {  //they have to swap
                        blocks.get(i).y--;
                        return;
                    }
```

```java
            //checks if there is an immovable block
            for (int i = noBlocks + 1; i < blocks.size(); i++)
                if (blocks.get(i).y == blocks.get(0).y)
                    if (blocks.get(i).x == blocks.get(0).x) { //found block
                        blocks.get(0).y--;  //agent goes back
                        return;
                    }
            break;
        }
    }

    //i assume input is valid and block is not over a start state or final state
    public void addImmovableBlock(int x, int y) {

        //it's over a start state
        if (x == size - 1)
            return;
        //it's over an end state
        if (y == 1 && x >= (size - noBlocks))
            return;

        blocks.add(new Block('i', x, y));
    }

    //used to test
    public void printNodes() {

        char[][] arr = new char[size][size];

        for (int i = 0; i < size; i++)
            for (int j = 0; j < size; j++)
                arr[i][j] = '0';

        for (int i = 0; i < blocks.size(); i++)
            arr[blocks.get(i).x][blocks.get(i).y] = blocks.get(i).name;

        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++)
                System.out.print(arr[i][j] + " ");
            System.out.println();
        }
        System.out.println();
    }

    public boolean isSolution() {

        int tmp = size - noBlocks;

        for (int i = 1; i <= noBlocks; i++)
            if (!(blocks.get(i).y == 1 && (blocks.get(i).name - 'A' + tmp) == blocks.get(i).x))
                return false;

        return true;
    }

    public ArrayList<Node> getFastMoves() {

        ArrayList<Node> moves = new ArrayList<Node>();
        Block agent = blocks.get(0);

        if ((agent.x > 0) && lastMove != 'D')
            moves.add(new Node(this, 'U'));

        if ((agent.y > 0) && lastMove != 'R')
            moves.add(new Node(this, 'L'));

        if ((agent.y < size - 1) && lastMove != 'L')
            moves.add(new Node(this, 'R'));

        if ((agent.x < size - 1) && lastMove != 'U')
```

```java
            moves.add(new Node(this, 'D'));

        return moves;
    }

    @SuppressWarnings("Duplicate")
    public ArrayList<Node> getMoves() {

        ArrayList<Node> moves = new ArrayList<Node>();
        Block agent = blocks.get(0);

        if (agent.x > 0)
            moves.add(new Node(this, 'U'));

        if (agent.y > 0)
            moves.add(new Node(this, 'L'));

        if (agent.y < size - 1)
            moves.add(new Node(this, 'R'));

        if (agent.x < size - 1)
            moves.add(new Node(this, 'D'));

        return moves;
    }

    public ArrayList<Node> getMovesObstacle() {

        ArrayList<Node> moves = new ArrayList<Node>();
        Block agent = blocks.get(0);

        if (agent.x > 0 && lastMove != 'D')
            moves.add(new Node(this, 'U', true));

        if (agent.y > 0 && lastMove != 'R')
            moves.add(new Node(this, 'L', true));

        if ((agent.y < size - 1) && lastMove != 'L')
            moves.add(new Node(this, 'R', true));

        if ((agent.x < size - 1) && lastMove != 'U')
            moves.add(new Node(this, 'D', true));

        return moves;
    }

    //cost to another node
    //used in case there is a different final state
    public int calculateCost(Node dest) {

        int cost = 0, costBlock;
        ArrayList<Block> a = this.getBlocks();
        ArrayList<Block> b = dest.getBlocks();
        for (int i = 1; i < a.size(); i++) {

            costBlock = Math.abs(a.get(i).x - b.get(i).x) + Math.abs(a.get(i).y - b.get(i).y);
            cost += costBlock;
        }

        cost += depth;
        this.cost = cost;
        return cost;
    }

    //cost to solution
    public int calculateCost() {

        int cost = 0, costBlock;
        ArrayList<Block> a = this.getBlocks();
```

```java
        for (int i = 1; i < a.size(); i++) {

            costBlock = Math.abs(a.get(i).x - (i + size - noBlocks - 1)) + Math.abs(a.get(i).y - 1);
            cost += costBlock;
        }

        cost += depth;
        this.cost = cost;
        return cost;
    }

    public ArrayList<Block> getBlocks() {
        return blocks;
    }

    public char getLastMove() {
        return lastMove;
    }

    public Node getParent() {
        return parent;
    }

    public int getSize() {
        return size;
    }

    public int getNoBlocks() {
        return noBlocks;
    }

    public int getCost() {
        return cost;
    }

    //used to test
    public void setBlock(int letter, int x, int y) {
        blocks.set(letter, new Block((char) ('A' + letter - 1), x, y));
    }

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

    public int getDepth() {
        return depth;
    }

    @Override
    public int compareTo(Object n) {
        Node tmp = (Node) n;
        return this.getCost() - tmp.getCost();
    }

    public int hashCode() {
        int sum = blocks.get(0).hashCode();
        for (Block i : blocks)
            sum = 7 * sum + i.hashCode();
        return sum;
    }
}
```

Main class

```java
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.util.*;
```

```java
//-Xmx16G jvm option needed/or at least more than default for size 4

public class Main {

    static BufferedWriter fout;
    static long nodes;
    static long fifteenMins = 1000 * 60 * 15, startTime;

    public static void main(String[] args) throws Exception {


        FileWriter fw = new FileWriter("out.txt");
        fout = new BufferedWriter(fw);
        Node s;

        boolean bfs = true, dfs = true, ids = true, astar = true;
        int j = 3; //default number of blocks

        //s=new Node(4,3);
        //s.printNodes();
        //no extras
        //increases the size until all search methods fail
        //to add extras change the name of the search methods by adding an prefixing
        //an i for when ther are immovable objects or f for graph search
        //so change bfs(s) to fbfs(s) or ibfs()
        //bfs=ids=astar=false;

        for (int i = 4; bfs||dfs||ids||astar ;i++){


            //s.addImmovableBlock( 0,i-1);
            s = new Node(i, j);

            fout.write('\n' + "Size is " + i + 'X' + i + '\n' + "Number of Blocks is " + j + '\n');

            //System.out.println(i);
            if (bfs)
                try {
                    bfs(s);
                } catch (Exception e) {
                    bfs = false;
                    fout.write("BFS failed because " + e.getMessage() + '\n');
                }
            if (dfs)
                try {
                    dfs(s);
                } catch (Exception e) {
                    dfs = false;
                    fout.write("DFS failed because " + e.getMessage() + '\n');
                }
            if (ids)
                try {
                    iterativeDeepening(s);
                } catch (Exception e) {
                    ids = false;
                    fout.write("IDS failed because " + e.getMessage() + '\n');
                }
            if (astar)
                try {
                    astar(s);
                } catch (Exception e) {
                    astar = false;
                    fout.write("A* failed because " + e.getMessage() + '\n');
                }
        }

        fout.close();
        fw.close();
    }
```

```java
    //breadth first search
    static void bfs(Node start) throws Exception {

        Queue<Node> q = new LinkedList<>();
        ArrayList<Node> moves;
        Node n;
        startTime = System.currentTimeMillis();
        nodes = 0;
        q.offer(start);

        while (!q.isEmpty()) {

            nodes++;
            n = q.poll();

            if ((System.currentTimeMillis() - startTime) > fiveMins)
                throw new Exception(" 15 minute exceeded ");

            if (n.isSolution()) {
                long time = System.currentTimeMillis() - startTime;
                fout.write('\n' + "BFS time = " + time + " ms" + '\n' + "Nodes expanded : " + nodes
+ '\n');
                printSol(foundSol(n));
                return;
            }

            moves = n.getMoves();
            for (Node c : moves) q.offer(c);
        }

    }

    //depth first search
    static void dfs(Node start) throws Exception {

        ArrayList<Node> moves;
        Node n = start;
        startTime = System.currentTimeMillis();
        nodes = 0;
        while (true) {

            nodes++;

            if ((System.currentTimeMillis() - startTime) > fiveMins)
                throw new Exception(" 15 minute exceeded ");

            if (n.isSolution()) {
                long time = System.currentTimeMillis() - startTime;
                fout.write('\n' + "DFS time = " + time + " ms" + '\n' + "Nodes expanded : " + nodes
+ '\n');
                //commented because the output file gets too big
                //printSol(foundSol(n));
                return;
            }

            moves = n.getMoves();
            Collections.shuffle(moves);
            n = moves.get(0);

        }
    }

    //iterative deepening
    static void iterativeDeepening(Node start) throws Exception {

        Node found;
        int depth = 0;
        startTime = System.currentTimeMillis();
```

```java
        nodes = 0;

        while (true) {

            found = dls(start, depth);

            if (found != null && found.isSolution()) {
                long time = System.currentTimeMillis() - startTime;
                fout.write('\n' + "IDS time = " + time + " ms" + '\n' + "Nodes expanded : " + nodes
+ '\n');
                printSol(foundSol(found));
                return;
            }

            depth++;

        }
    }

    static Node dls(Node n, int depth) throws Exception {

        nodes++;
        ArrayList<Node> moves;
        Node found;

        if ((System.currentTimeMillis() - startTime) > fiveMins)
            throw new Exception(" 15 minute exceeded ");

        //n is solution
        if (depth == 0 && n.isSolution())
            return n;

        if (depth > 0) {

            moves = n.getMoves();

            for (Node node : moves) {

                found = dls(node, depth - 1);

                //a solution was found, stopping the search
                if (found != null)
                    return found;
            }
        }
        return null;
    }

    //A*
    static void astar(Node start) throws Exception {

        ArrayList<Node> moves;
        Queue<Node> q = new PriorityQueue<>();
        Node tmp;
        nodes = 0;
        startTime = System.currentTimeMillis();
        start.setDepth(0);
        start.calculateCost();
        q.offer(start);
        while (!q.isEmpty()) {

            tmp = q.poll();
            nodes++;

            if ((System.currentTimeMillis() - startTime) > fiveMins)
                throw new Exception(" 15 minute exceeded ");

            if (tmp.isSolution()) {
                long time = System.currentTimeMillis() - startTime;
                fout.write('\n' + "A* time = " + time + " ms" + '\n' + "Nodes expanded : " + nodes
```

```java
+ '\n');
                printSol(foundSol(tmp));
                return;
            }

            moves = tmp.getMoves();

            for (Node node : moves) {

                node.setDepth(tmp.getDepth() + 1);
                node.calculateCost();
                q.offer(node);

            }
        }
    }

    //improved getMoves and graph search bfs
    static void fbfs(Node start) throws Exception {


        //needed for graph search
        HashSet<Node> uniqueNodes = new HashSet<>();
        uniqueNodes.add(start);
        Queue<Node> q = new LinkedList<>();
        ArrayList<Node> moves;
        Node n;
        startTime = System.currentTimeMillis();
        nodes = 0;
        q.offer(start);

        while (!q.isEmpty()) {

            nodes++;
            n = q.poll();

            if ((System.currentTimeMillis() - startTime) > fiveMins)
                throw new Exception(" 15 minute exceeded ");

            if (n.isSolution()) {
                long time = System.currentTimeMillis() - startTime;
                fout.write('\n' + "BFS time = " + time + " ms" + '\n' + "Nodes expanded : " + nodes
+ '\n');
                printSol(foundSol(n));
                return;
            }

            moves = n.getFastMoves();

            for (Node c : moves)
                if (!uniqueNodes.contains(c) && !q.contains(c)) {     //graph search

                    q.offer(c);
                    uniqueNodes.add(c);
                }

        }

    }

    //improved getMoves and graph search dfs
    static void fdfs(Node start) throws Exception {

        //for graph search
        HashSet<Node> uniqueNodes = new HashSet<>();
        uniqueNodes.add(start);
        Stack<Node> stack = new Stack<>();
        ArrayList<Node> moves;
        Node n;
        startTime = System.currentTimeMillis();
```

```
        nodes = 0;
        stack.push(start);

        while (!stack.isEmpty()) {

            nodes++;
            n = stack.pop();

            if ((System.currentTimeMillis() - startTime) > fiveMins)
                throw new Exception(" 15 minute exceeded ");

            if (n.isSolution()) {
                long time = System.currentTimeMillis() - startTime;
                fout.write('\n' + "DFS time = " + time + " ms" + '\n' + "Nodes expanded : " + nodes
+ '\n');
                //commented because the output file gets too big
                //printSol(foundSol(n));
                return;
            }

            moves = n.getFastMoves();
            Collections.shuffle(moves);
            for (Node c : moves)
                if (!uniqueNodes.contains(c)) {   //graph search
                    stack.push(c);
                    uniqueNodes.add(c);
                }
        }
    }

    //improved getMoves and graph search A*
    static void fastar(Node start) throws Exception {

        ArrayList<Node> moves;
        Queue<Node> q = new PriorityQueue<>();
        //for graph search
        //upon further tests i found out it slowed the search
        //HashSet<Node> uniqueNodes = new HashSet<>();
        //uniqueNodes.add(start);
        Node tmp;
        nodes = 0;
        startTime = System.currentTimeMillis();
        start.setDepth(0);
        start.calculateCost();
        q.offer(start);

        while (!q.isEmpty()) {

            tmp = q.poll();
            nodes++;

            if ((System.currentTimeMillis() - startTime) > fiveMins)
                throw new Exception(" 15 minute exceeded ");

            if (tmp.isSolution()) {
                long time = System.currentTimeMillis() - startTime;
                fout.write('\n' + "A* time = " + time + " ms" + '\n' + "Nodes expanded : " + nodes
+ '\n');
                printSol(foundSol(tmp));
                return;
            }

            moves = tmp.getFastMoves();

            for (Node node : moves)
                if (!q.contains(node))   //for graph search
                {
                    node.setDepth(tmp.getDepth() + 1);
                    node.calculateCost();
                    q.offer(node);
```

```
                    //uniqueNodes.add(node);
                }

        }
    }

    //breadth first search default with immovable objects
    @SuppressWarnings("Duplicates")
    static void ibfs(Node start) throws Exception {

        //needed for graph search
        HashSet<Node> uniqueNodes = new HashSet<>();
        uniqueNodes.add(start);
        Queue<Node> q = new LinkedList<>();
        ArrayList<Node> moves;
        Node n;
        startTime = System.currentTimeMillis();
        nodes = 0;
        q.offer(start);

        while (!q.isEmpty()) {

            nodes++;
            n = q.poll();

            if ((System.currentTimeMillis() - startTime) > fiveMins)
                throw new Exception(" 15 minute exceeded ");

            if (n.isSolution()) {
                long time = System.currentTimeMillis() - startTime;
                fout.write('\n' + "BFS time = " + time + " ms" + '\n' + "Nodes expanded : " + nodes
+ '\n');
                printSol(foundSol(n));
                return;
            }

            moves = n.getMovesObstacle();

            for (Node c : moves)
                if (!uniqueNodes.contains(c) && !q.contains(c)) {     //graph search

                    q.offer(c);
                    uniqueNodes.add(c);
                }

        }

    }

    //depth first search default with immovable objects
    @SuppressWarnings("Duplicates")
    static void idfs(Node start) throws Exception {

        //for graph search
        HashSet<Node> uniqueNodes = new HashSet<>();
        uniqueNodes.add(start);
        Stack<Node> stack = new Stack<>();
        ArrayList<Node> moves;
        Node n;
        startTime = System.currentTimeMillis();
        nodes = 0;
        stack.push(start);

        while (!stack.isEmpty()) {

            nodes++;
            n = stack.pop();

            if ((System.currentTimeMillis() - startTime) > fiveMins)
                throw new Exception(" 15 minute exceeded ");
```

```java
            if (n.isSolution()) {
                long time = System.currentTimeMillis() - startTime;
                fout.write('\n' + "DFS time = " + time + " ms" + '\n' + "Nodes expanded : " + nodes
+ '\n');
                //commented because the output file gets too big
                printSol(foundSol(n));
                return;
            }

            moves = n.getMovesObstacle();

            Collections.shuffle(moves);
            for (Node c : moves)
                if (!uniqueNodes.contains(c)) {   //graph search
                    stack.push(c);
                    uniqueNodes.add(c);
                }
        }
    }


    //A* default with immovable objects
    @SuppressWarnings("Duplicates")
    static void iastar(Node start) throws Exception {

        ArrayList<Node> moves;
        Queue<Node> q = new PriorityQueue<>();
        //for graph search
        HashSet<Node> uniqueNodes = new HashSet<>();
        uniqueNodes.add(start);
        Node tmp;
        nodes = 0;
        startTime = System.currentTimeMillis();
        start.setDepth(0);
        start.calculateCost();
        q.offer(start);

        while (!q.isEmpty()) {

            tmp = q.poll();
            nodes++;

            if ((System.currentTimeMillis() - startTime) > fiveMins)
                throw new Exception(" 15 minute exceeded ");

            if (tmp.isSolution()) {
                long time = System.currentTimeMillis() - startTime;
                fout.write('\n' + "A* time = " + time + " ms" + '\n' + "Nodes expanded : " + nodes
+ '\n');
                printSol(foundSol(tmp));
                return;
            }

            moves = tmp.getMovesObstacle();

            for (Node node : moves)
                if (!uniqueNodes.contains(node) && !q.contains(node))   //for graph search
                {
                    node.setDepth(tmp.getDepth() + 1);
                    node.calculateCost();
                    q.offer(node);
                    uniqueNodes.add(node);
                }

        }
    }

    //auxiliary functions
```

```java
    static void printSol(Stack<Character> solution) throws Exception {
        fout.write("Solution path is : ");
        while (!solution.isEmpty())
            fout.write(solution.pop() + " ");
        fout.write('\n');
    }

    //puts the solution in a stack
    //as it is written end to beginning
    //and stack is lifo
    static Stack<Character> foundSol(Node n) {

        Stack<Character> solution = new Stack<>();
        while (n.getParent() != null) {
            solution.add(n.getLastMove());
            n = n.getParent();
        }
        return solution;
    }

}
```

7. Output for 4X4 and 3 blocks

```
Size is 4X4
Number of Blocks is 3

BFS time = 48446 ms
Nodes expanded : 5423596
Solution path is : U L L D L U R D R U U L D L

DFS time = 7 ms
Nodes expanded : 4196
Solution path is : L R L L L U D U R U L R D U U D L D U D D U U R U R D D R U L U L R L L R R D R
L L R U D U L L R D D L R D R L U U R L R R L R L D R L U L U L R L D R D L R D U R R U U L D U D L
L U R R D R D D U D U U U D D L R D L U L D U R D R L R L R L U L R L L R U D R U D U U R L R L L R
D R L U R L L D R L R L U L R R L R D L D L D R L R R U R D L L L U U D U R D U R D U U R D D U D L
R D L L L U U U D U R D R D L D U L R L R R U R L L U D L U R L R L R D D L U D R U U D L U R L R D
U R R D L U D L D U L D U U R R L L D D R D L U D U U D D R U L D U D U D U U U D R R D L U D D R U
R L L D R R U U U L R D U L D D U U D D U D R D L U R L L D R U R U D L D L U D R R U L L L D U U D
R R R L R D U L L D U D R L U L U R U R R D L U D U D L U R L R D D R L L U L R U D D D R L L U U U
D U D D D U D R R R U U L L U R R D D U U L L D R L U D R U R D D L R D L U R D U D U U U D D L D L
R L U U D R U U D R L L D D U R U L D D R U L U U L R R D R L D U R D U U L D U L R L L R R L D U R
D R D U L R U L L R L L D R D R L U U L R R L D L R L D D R R R L U D U L L D U R U L U D U R R R L
R L D L L U R D L R L D R L D R U U D U R U R D L D L D R L L U R R L L U U D U R L R L D U R D U R
R L D R U D D L D U D R L U U L L U R L D D U D D U R R R D L R L U U D L U D L D U U R R R D D U U
U L L D D U U L R L R R D U L R D L R U D D D U R L L D L U D U R L D R L R L R R R L L R L L R R U
D L L U D R L R R L L U U D D U R R R L L D U D R U R D U D U L L U D U U R R D L L D U L D R U L R
D R U D R L D L L R U L U U D D D U U D U U D R U L D U D D U U R R D L U D L U D R U D L R L R D U
L D U U R D D U R R L U L L R L R L D R R L R D D R U U L R D L D L L U U U R L R D L U R L D D U U
R R D R L D D L R L L R L R L R L U R R D L L R L U U R L R U R D L U D U L D D U U R L R L R D R D
U L D L U D D U R U D D U L R D R R L U R U U L R D D D L R U L D L L R R L L U D U R R R U U L R L
D U D D U U D R D U D U D D U D U U U L R L R D D L R L R L L R U U R L D L D D L U R L R R R L U R
U D L L U D L D D R R L U R R L R D U D U D U U D U L U R L D D R U U D D L L U R U R L L R L R L D
R L U D R R D D L L R R U U U L L R R L L L R D U D L U D R D D L R U D U R D L L U R D L U R L R R
D U U L L R R L R D R L D L L R U D U L U D D R L U R L D R U U L D D R R L U D U U R L R L D D L U
D R U L R D L R U D U R D U L D R L R U U D D R L R U U L L R U R L L D L R R U D D D R L R L R L L
L U D U U U D R L U D D U U R L D D D U R L U U D U R D D L U U D R R R U D L L U D D R U R D D U L
D R L U R D L U D U L U U R L R L L R R D U D R D L R L L R L D U L R D L U U D D R R R U U L R U D
L R D U U L D L D R L L U R R R L U L D R U L L R L D D U R L D D R U L D U D R R L L R L R U R R U
D L U R L R U L L D U L R D D L R R D U R L D R L L L U D R U U D L R R D R U D L U D L L U U R U D
R U D R U L D R L D D R L R L U U L L D D R R R L U L L D U D U U U R R R L R D U L D D U U R L R D
L U R L R D D L U U D D R U D U L D D L U U R D L D R R U U D D L U L U R L L R U D L R U R R D U L
L L R R L D L U R D D D L R U D U U D D U U D U D U R L U R R L R D L L D U U R L L D U D D R U R U
```

```
L D L U D U R R D D D U L D R L L R R L U U U D R L D U L R U L D D U U R D L U R R R L D U R D L D
D R L R U D L L U R L U L D R U D U R L D R D U D L R U U D R L L D R R U L U R U D D L R D L U U R
L L R U R D U D L D U D R U D D U D U U U L D R D D L R L L U L R L D R U D U U R R L U D D R L R U
U D D D U U L L U D L U D U D U D R R L D D U U D U R D U R U D D L U R U D L R D U D D L R U L R L
L U L D U U D D D U D R R R L R U D L R U L U L L U D D D R U R D R U L D U L L D R R U U L L R L D
R L R U L U D D U D U R U D D R R L L D U D R L R R L L L R U R D L U R R D L U U L R U D D U L L R
U R L L R R D U R D D U D U L R L D D L R R L R U U D D L R U U L L U R R D U D U D U D L L D U D R
U D R D L L R L R R L R L U R L U L D D L R L U R L D U D U D R L R R R L U R L L L D R L R R R U U
U D U D U L R D D D L L R R L R U L L U U L D D R R R D U L U D R L R U U D D D L L U L U R R L D R
U U D U R L L R L R D R D L U U L L R L R R R D L D D L L R U R R L R L U U D U L D U R D D L D L U
U R L U D R U L R L R L D U R R D D L D U D L R R R U D U D U U U D D U L L L U D U R L R L R R L R
L R D L R U R D L U R L R L D R U D U D U L D U D D L D U L R L U R D L U R U D U R D L D D R L U R
L R L D R L R U D L U D R R L U U U D U R D D D L R U L D R U D L U U U R L L D D R D R L R L U U L
L R L R L D D R L U U U R D U D R D L L R U D D R R U L R U U D L D L U R U R L L D D D R L L U R D
L U R L R R R D U D U D L R U U D U L U R L D L R U D L D U R R U L D L D R U U R D U L L D R L U L
R L D R L D R L R U D D U R U L U D R U D R U L L D U D R U D U D L R D D L L U U D D R R U U D R L
L U L R D U U R L R D U D L D L U R D U U L D R R D R U U D L D R L R L L D L R R U L R D U L U D U
L R R U D U L L R L R L D U R D D U L U D D U U D D U U R L D U R L D D U U R L R L R R R L R D U D
L R L R U L L D R U R L L D D U D L U U D D R D R R U L R D L R L R U D L U U U D R L D L L D U R R
R U U D D L L D R L U D U U U R D R U D D U D L R U L U L D L D D U R D R U L U L U D D R D R R L L
L R U U U R D R L L R R L L U D L U R D D L D U U D R L R R D R U L R D L U D L L R U R D R L L R R
U L U R L L L R L R D R D R U D U D U U U D U L R D U L L R R L R D D U U D L U R D D U U D U L D U
R L L D R U R D D D U U L R D D L U L D R U D L L U U R D R D U R D L R U L D L L U D R L U U D U U
D D U U R D R R D D U U U L D U D R U D L D R D L L U D R U U D D L R U D L L R L R R U R U D L L U
U L R R D D U R U D U L L L D D D R U R R D L L U R U U D L D D L U D U D R R U L R D L U D R R U U
D D L R L L L U D U D U D U R U R D U R L U D D R D U L U U L D R D R D U D U D U L D U R U L R D L
R U L L U L D U D D U D U R D U U D U R R D U D D U L D U R L U D D L L U D D U U R R U L R L R R L
D U R L D R U D L D U D U U D R U D D D U U D L D R U D U U D U L D U R L U R L R L D D R U U D U D
D U U L R L D D D L R L U L R U R D R U D U L L D D L U D R U L R R D U L D L U D R R U L D U D U R
D U L L D U U R U D R R U L D D U U D L R D D L R L L R U D R U U L L R L U D D D R R R L R U L L D
L U D U D U U U D D U D D R L U D U D R U R L R L D R L U R D L U R D R U L R D U L R D L R U L R L
D L R R L L R R L R L R U U L U D U R L R D L D U R U D L L U L D D U U R L R L D D R R D L R U L D
U U R D D L U U L D D R U U L R R U L D R L R D D L L R R U R L D L U D U L U R D U D U L D R D R R
L R U D L L R R U U L U R D U L R D D U D D U U L R D U L D L L R D L U U U R L D R U L D U R R D D
U D R D U U L L U D D U L R R U L R D L U R R L L L D U D R R L R L L U R L R R R D L U L L D U D D
D U D U U U D U R L R R R D D U D L R L D U D L R R L U L L D R L U D R U R R D L R L L R R U L U R
D U L L L R R D U R U L D U L L R R L L D R L U D R L D D R R R U D U U D L R D L R U D L R L R L U
R D L R L R U L U D D R U D U L L D R R U U D D U L U U L R L R D D D L L R U U U R L R R D U L D R
D U D U U D D U D L R D L R L R L U U U R D D D U D U L U L U L D U D R R L D L D R R L R U U U R D
L R U D L L U R D L U R D L L R U R L D D D L R L R R L R L U D L U R D L R L R L U R D U D R L R R
U D L U U U R D U L L D L R U R D U R D L U D L R D L U D R D R L U D L U R R D U U L U R D L U L L
D U D D D R L R U R D L U U U D R L D D R L U R L L D U R D R U U L U D R R L L R L U L R D D R D U
L R R L D L R U R U U D L L R U D L R R L D L R D L R L R R U U D U D L R U U L L R L D U L D R R R
L D D L U U L D U U R R D R U D U L L R L L D D D R L R U L R D L R L U U D D U D R L U U D D R R R
L R L L R R L U U L L R D D L U U D D U U R D U D R U R L R U L R L R D L U L D U L D U R L R D R R
D L R D L L U L R R R U U L L D R U D D L R L R D L R R U D U U D D L U R D U U L L R D R U D D U L
R U L L L D U R R L U D U R R L R D U L L L R L R D L U R D L U D U D D U R D L R R R D L L L U R U
D D R L R R U U D L L U D D R R L L R R L R L R U L L D U D U U D D U R D L U U R L L D U R R D U L
R R L R D U D D U U D D L L R U R U U L R D U D D U U D D L U U R L L D R U L R L L R D U L D D U R
D R R D L R U D L L L U U R L D R D R L U U L R D L R U R D R L D L U R L U D U U R D R U L D U D R
D U L D D U R L R D L U R D U L D U U U D L U L R D R U D L D U U D R L U D L U D D D R U L R R R L
L D R L U D R L U D L U D U R R U U D U R D D L L U R U D L U R D U L R L L D D D R U L D R L U R R
U R D D U U D D U L D R U L L L D U U U R L D D U D R R R D U L U R D L D R U U U D D L R L R L R L
L D U D R

IDS time = 3527 ms
Nodes expanded : 7849127
Solution path is : U L L D L U R D R U U L D L

A* time = 32 ms
Nodes expanded : 20421
Solution path is : U L L D L U R D R U U L D L
```