A* LABORATORY PROJECT

ANTONIO TRUJILLO
ALBERTO TRIGUEROS
JESÚS FUENTES
MARÍA PEINADO
GROUP C

INDEX

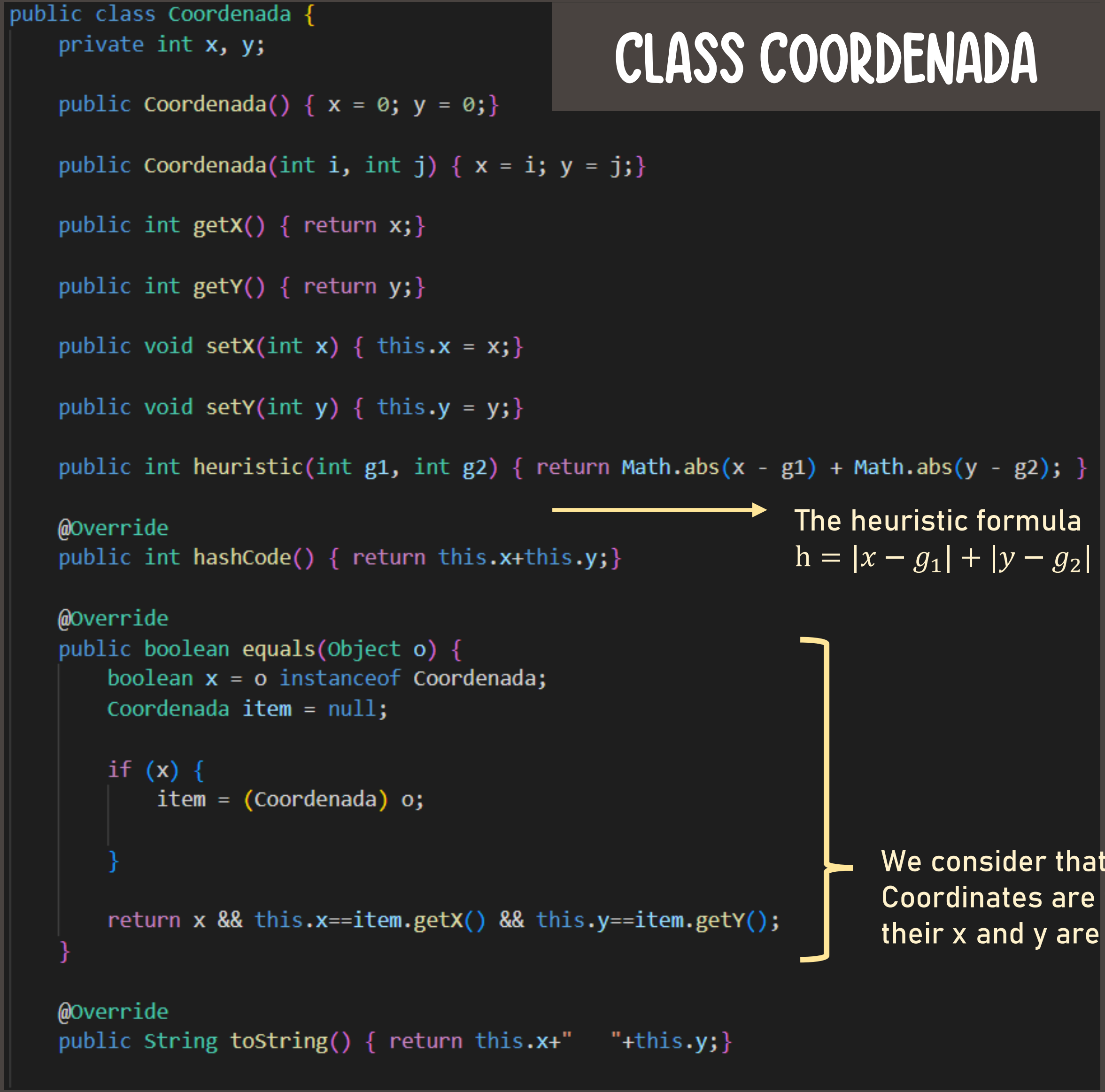
1. Introduction
2. Structure of the application, design choices
3. Experimental results, performance slots, coments.

INTRODUCTION

This project is focused on creating a function with the ability of solving a maze. The algorithm should create a path from the initial point to the end, avoiding obstacles and staying into the maze. Furthermore, it should alert if the initial and end point are in the same place or if one of these points match with an obstacle. In the following slides, we are going to explain how we got that.

STRUCTURE OF THE APPLICATION, DESIGN CHOICES

CLASS COORDENADA



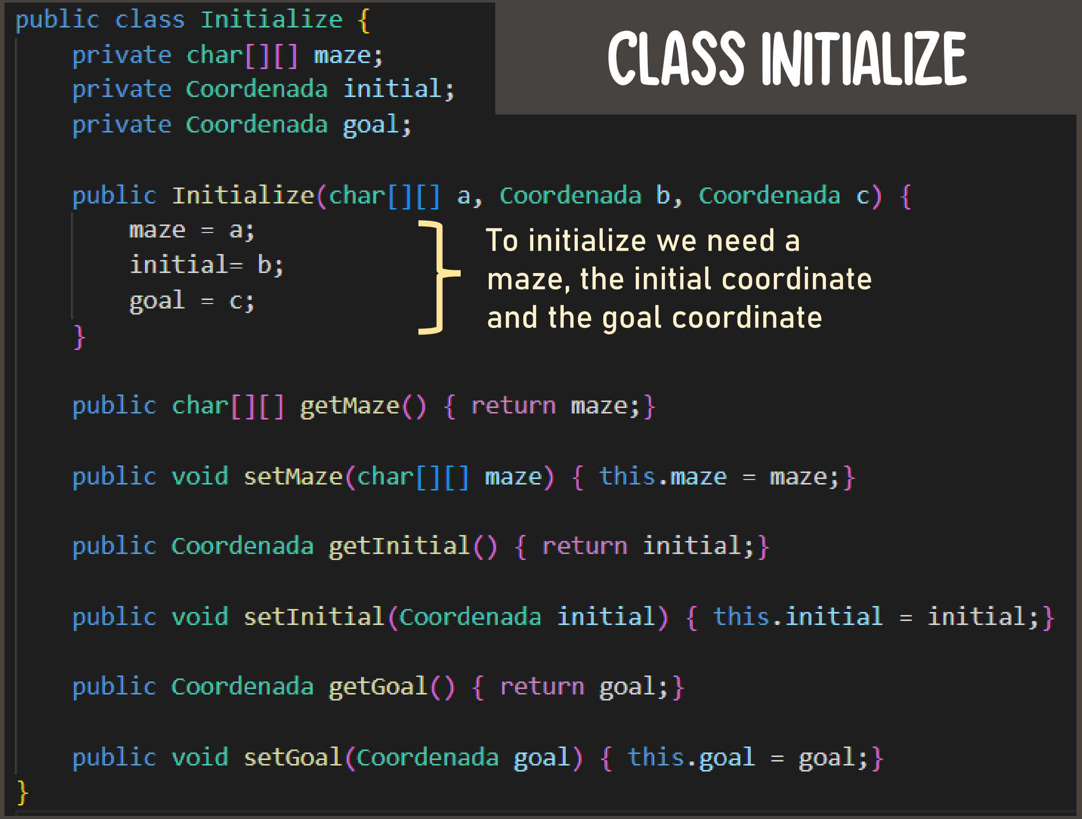
```java
public class Coordenada {
    private int x, y;

    public Coordenada() { x = 0; y = 0;}

    public Coordenada(int i, int j) { x = i; y = j;}

    public int getX() { return x;}

    public int getY() { return y;}

    public void setX(int x) { this.x = x;}

    public void setY(int y) { this.y = y;}

    public int heuristic(int g1, int g2) { return Math.abs(x - g1) + Math.abs(y - g2); }

    @Override
    public int hashCode() { return this.x+this.y;}

    @Override
    public boolean equals(Object o) {
        boolean x = o instanceof Coordenada;
        Coordenada item = null;

        if (x) {
            item = (Coordenada) o;

        }

        return x && this.x==item.getX() && this.y==item.getY();
    }

    @Override
    public String toString() { return this.x+"   "+this.y;}
```

The heuristic formula

$$h = |x - g_1| + |y - g_2|$$

We consider that two Coordinates are the same if their x and y are equal.

CLASS INITIALIZE



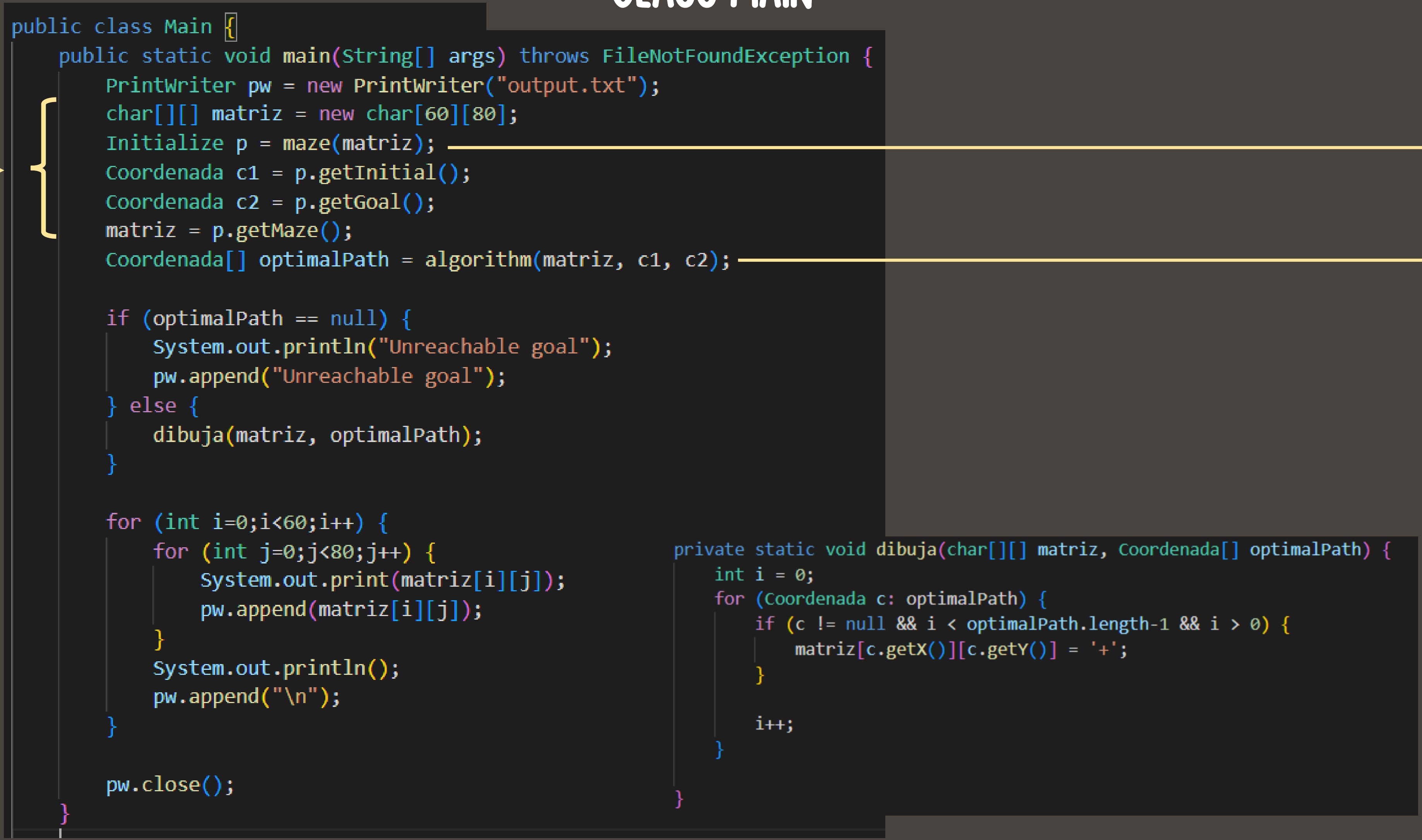
```java
public class Initialize {
    private char[][] maze;
    private Coordenada initial;
    private Coordenada goal;

    public Initialize(char[][] a, Coordenada b, Coordenada c) {
        maze = a;
        initial= b;
        goal = c;
    }

    public char[][] getMaze() { return maze;}

    public void setMaze(char[][] maze) { this.maze = maze;}

    public Coordenada getInitial() { return initial;}

    public void setInitial(Coordenada initial) { this.initial = initial;}

    public Coordenada getGoal() { return goal;}

    public void setGoal(Coordenada goal) { this.goal = goal;}
}
```

To initialize we need a maze, the initial coordinate and the goal coordinate

STRUCTURE OF THE APPLICATION, DESIGN CHOICES

CLASS MAIN

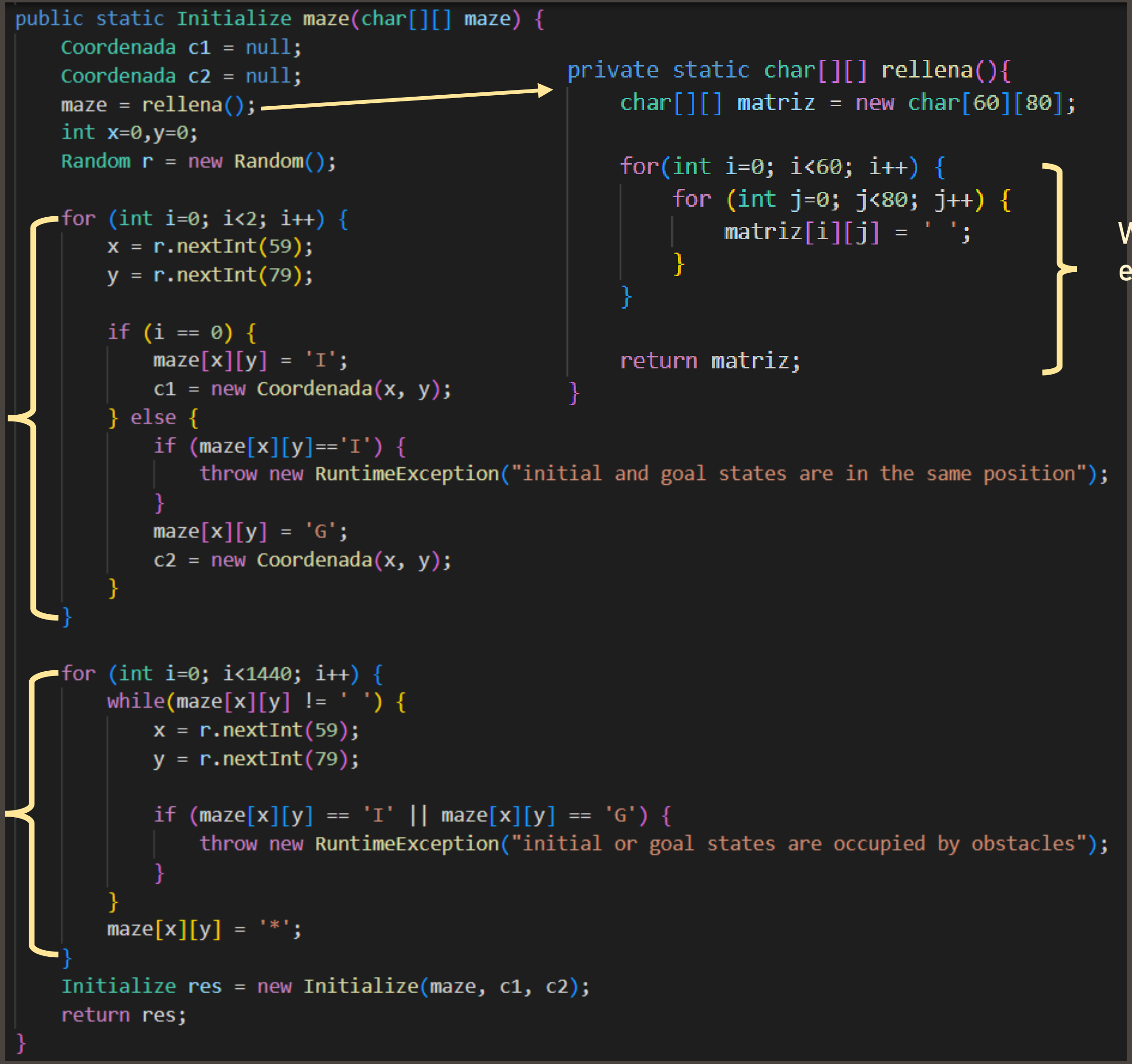
We create an empty maze 60x80 and we initialize by using the function maze.



```java
public class Main {
    public static void main(String[] args) throws FileNotFoundException {
        PrintWriter pw = new PrintWriter("output.txt");
        char[][] matriz = new char[60][80];
        Initialize p = maze(matriz);
        Coordenada c1 = p.getInitial();
        Coordenada c2 = p.getGoal();
        matriz = p.getMaze();
        Coordenada[] optimalPath = algorithm(matriz, c1, c2);

        if (optimalPath == null) {
            System.out.println("Unreachable goal");
            pw.append("Unreachable goal");
        } else {
            dibuja(matriz, optimalPath);
        }

        for (int i=0;i<60;i++) {
            for (int j=0;j<80;j++) {
                System.out.print(matriz[i][j]);
                pw.append(matriz[i][j]);
            }
            System.out.println();
            pw.append("\n");
        }

        pw.close();
    }
```

```java
private static void dibuja(char[][] matriz, Coordenada[] optimalPath) {
    int i = 0;
    for (Coordenada c: optimalPath) {
        if (c != null && i < optimalPath.length-1 && i > 0) {
            matriz[c.getX()][c.getY()] = '+';
        }

        i++;
    }

}
```

STRUCTURE OF THE APPLICATION, DESIGN CHOICES



```java
public static Initialize maze(char[][] maze) {
    Coordenada c1 = null;
    Coordenada c2 = null;
    maze = rellena();
    int x=0,y=0;
    Random r = new Random();

    for (int i=0; i<2; i++) {
        x = r.nextInt(59);
        y = r.nextInt(79);

        if (i == 0) {
            maze[x][y] = 'I';
            c1 = new Coordenada(x, y);
        } else {
            if (maze[x][y]=='I') {
                throw new RuntimeException("initial and goal states are in the same position");
            }
            maze[x][y] = 'G';
            c2 = new Coordenada(x, y);
        }
    }

    for (int i=0; i<1440; i++) {
        while(maze[x][y] != ' ') {
            x = r.nextInt(59);
            y = r.nextInt(79);

            if (maze[x][y] == 'I' || maze[x][y] == 'G') {
                throw new RuntimeException("initial or goal states are occupied by obstacles");
            }
        }
        maze[x][y] = '*';
    }
    Initialize res = new Initialize(maze, c1, c2);
    return res;
}
```

```java
private static char[][] rellena(){
    char[][] matriz = new char[60][80];

    for(int i=0; i<60; i++) {
        for (int j=0; j<80; j++) {
            matriz[i][j] = ' ';
        }
    }

    return matriz;
}
```

We locate the initial and the goal coordinate by using the class random and making sure that they are different.

We fill the 1/3 of the maze of obstacles (taking care of not putting one of them in the initial or the goal coordinate).

We fill the matrix with empty space.

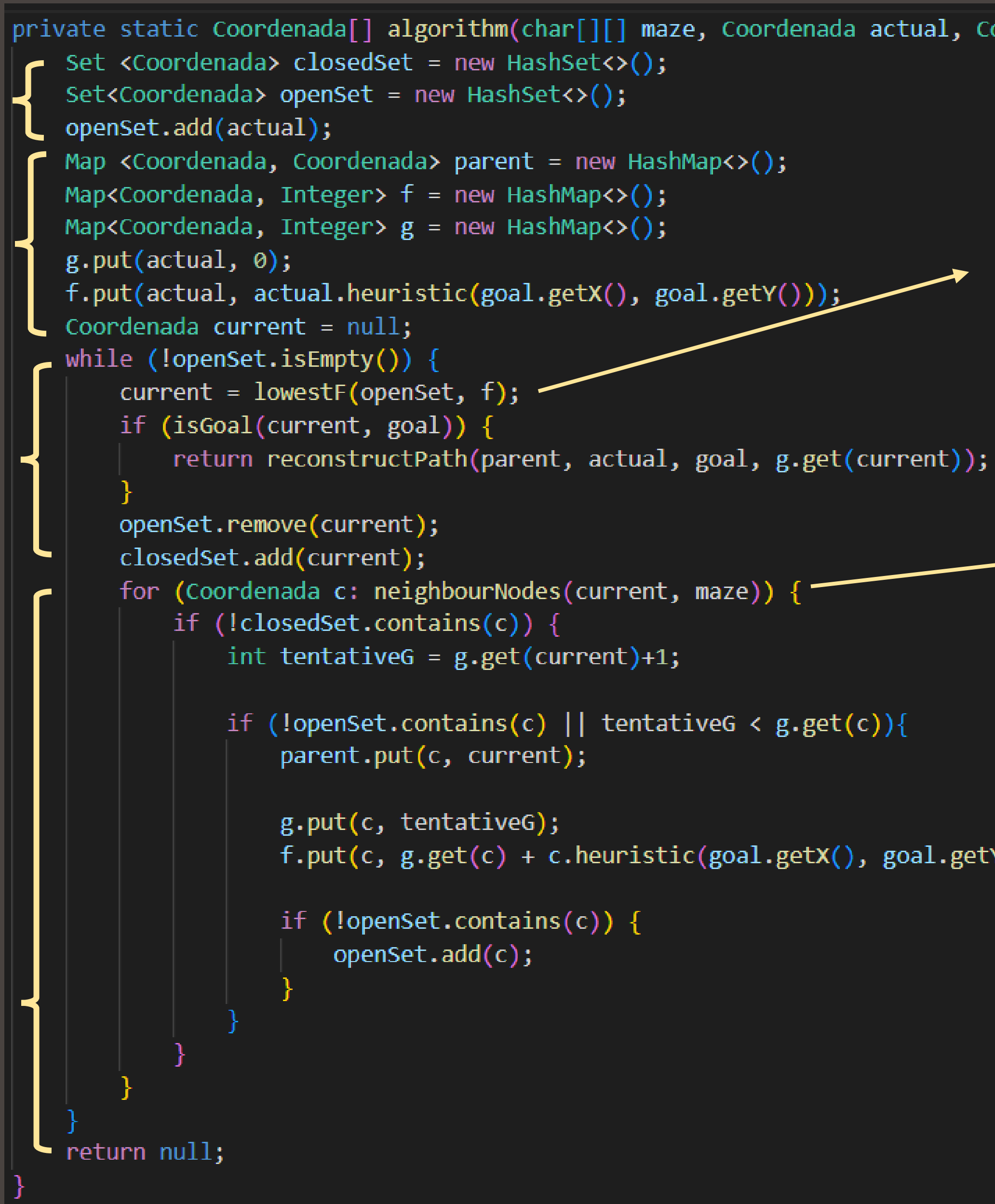
STRUCTURE OF THE APPLICATION, DESIGN CHOICES

We create the closedSet and the openSet. We add to the openSet the initial.

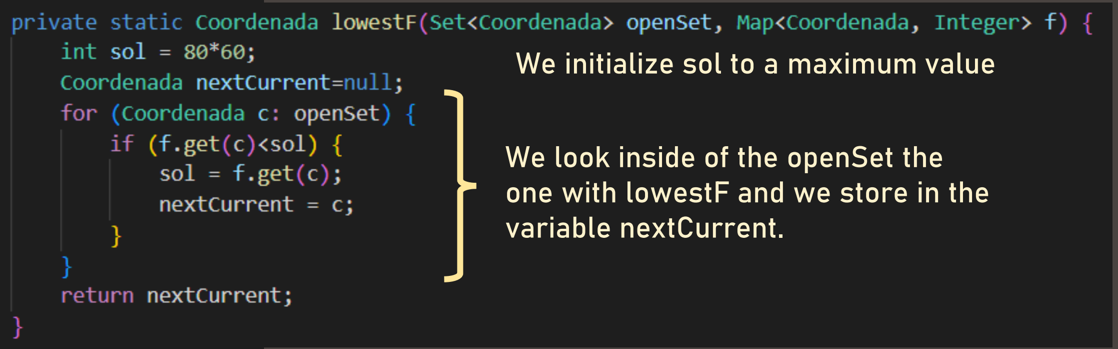
We use Map for the path, for f and for g.
In g we add the initial and the value 0.
In f we add the initial with the heuristic.

While the openSet is empty we are searching the lowesF, if we obtain the goal we return the path.
We remove from the openSet the node we are studying and we added to the closed set.

We study all de neighbourNodes that are still in the openSet or even not in there, changing it g if is necessary and including it f in the Map, and finally adding to the openset if they were not there.



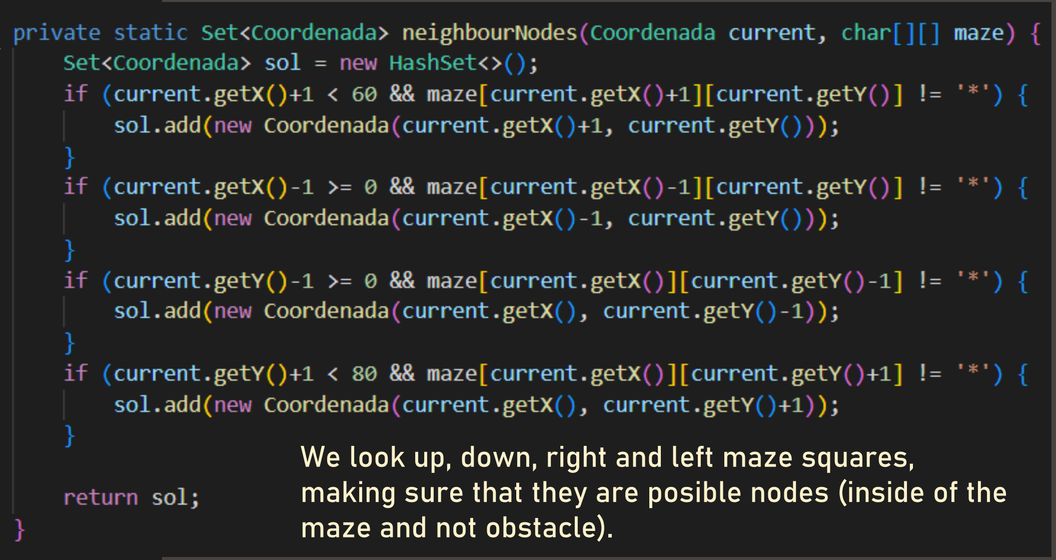
```java
private static Coordenada[] algorithm(char[][] maze, Coordenada actual, Coordenada goal) {
    Set <Coordenada> closedSet = new HashSet<>();
    Set<Coordenada> openSet = new HashSet<>();
    openSet.add(actual);
    Map <Coordenada, Coordenada> parent = new HashMap<>();
    Map<Coordenada, Integer> f = new HashMap<>();
    Map<Coordenada, Integer> g = new HashMap<>();
    g.put(actual, 0);
    f.put(actual, actual.heuristic(goal.getX(), goal.getY()));
    Coordenada current = null;
    while (!openSet.isEmpty()) {
        current = lowestF(openSet, f);
        if (isGoal(current, goal)) {
            return reconstructPath(parent, actual, goal, g.get(current));
        }
        openSet.remove(current);
        closedSet.add(current);
        for (Coordenada c: neighbourNodes(current, maze)) {
            if (!closedSet.contains(c)) {
                int tentativeG = g.get(current)+1;

                if (!openSet.contains(c) || tentativeG < g.get(c)){
                    parent.put(c, current);

                    g.put(c, tentativeG);
                    f.put(c, g.get(c) + c.heuristic(goal.getX(), goal.getY()));

                    if (!openSet.contains(c)) {
                        openSet.add(c);
                    }
                }
            }
        }
    }
    return null;
}
```



```java
private static Coordenada lowestF(Set<Coordenada> openSet, Map<Coordenada, Integer> f) {
    int sol = 80*60;
    Coordenada nextCurrent=null;
    for (Coordenada c: openSet) {
        if (f.get(c)<sol) {
            sol = f.get(c);
            nextCurrent = c;
        }
    }
    return nextCurrent;
}
```

We initialize sol to a maximum value

We look inside of the openSet the one with lowestF and we store in the variable nextCurrent.



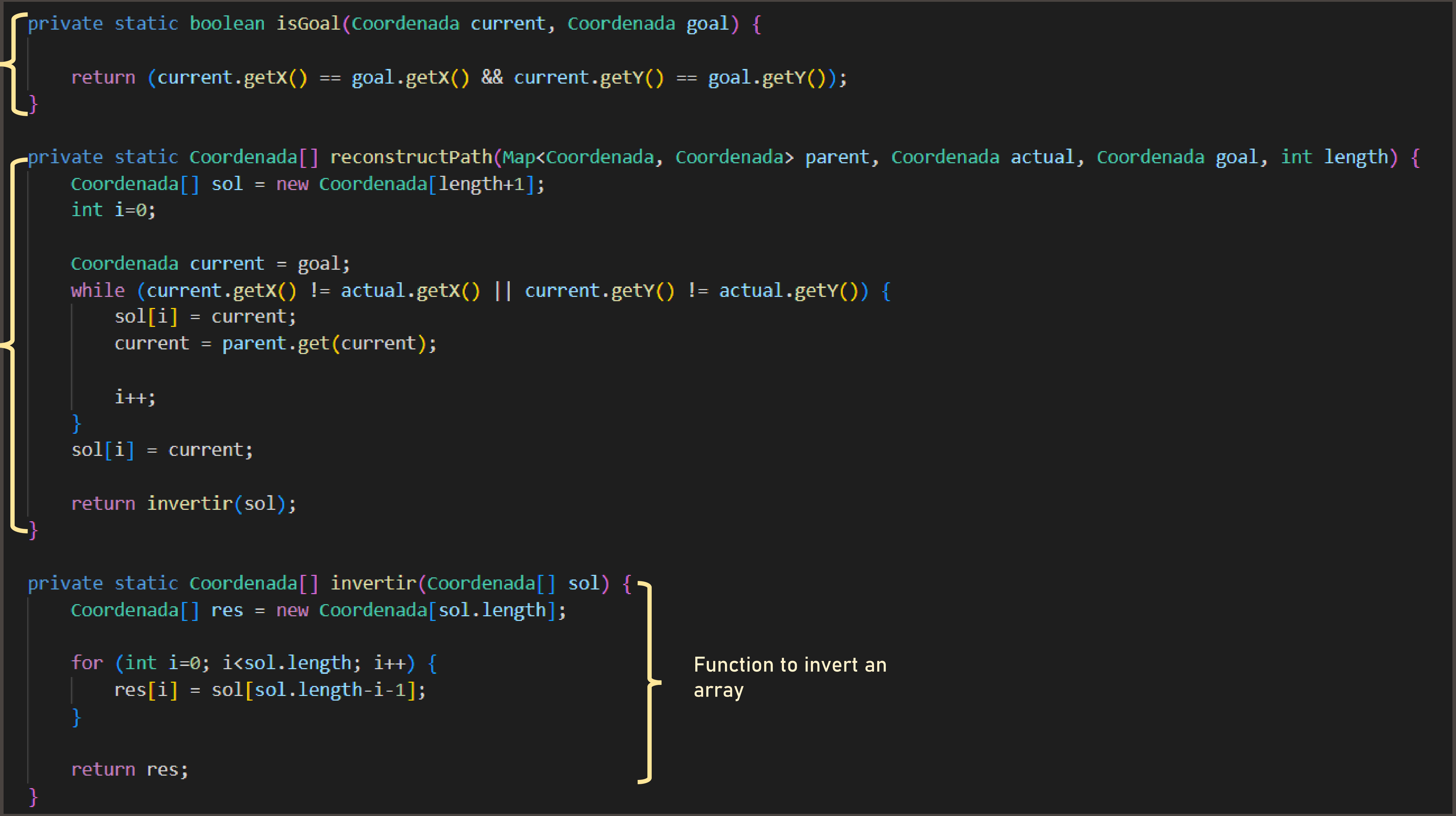
```java
private static Set<Coordenada> neighbourNodes(Coordenada current, char[][] maze) {
    Set<Coordenada> sol = new HashSet<>();
    if (current.getX()+1 < 60 && maze[current.getX()+1][current.getY()] != '*') {
        sol.add(new Coordenada(current.getX()+1, current.getY()));
    }
    if (current.getX()-1 >= 0 && maze[current.getX()-1][current.getY()] != '*') {
        sol.add(new Coordenada(current.getX()-1, current.getY()));
    }
    if (current.getY()-1 >= 0 && maze[current.getX()][current.getY()-1] != '*') {
        sol.add(new Coordenada(current.getX(), current.getY()-1));
    }
    if (current.getY()+1 < 80 && maze[current.getX()][current.getY()+1] != '*') {
        sol.add(new Coordenada(current.getX(), current.getY()+1));
    }

    return sol;
}
```

We look up, down, right and left maze squares, making sure that they are posible nodes (inside of the maze and not obstacle).

STRUCTURE OF THE APPLICATION, DESIGN CHOICES

Return if the current coordinate is the same as the goal one

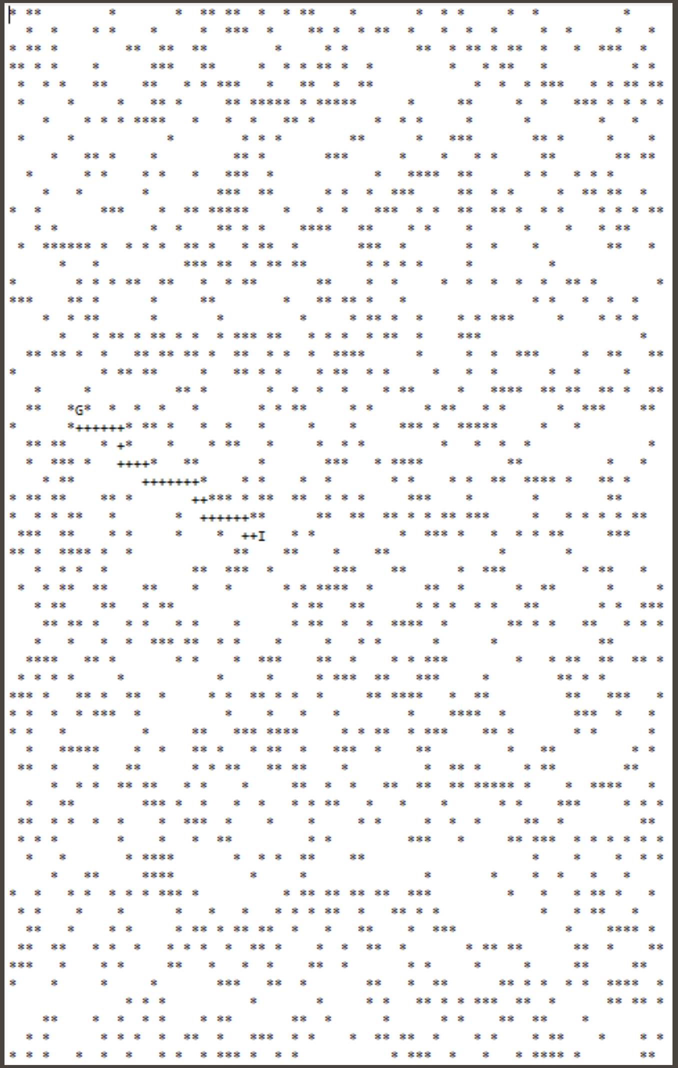
The path solution is an array of coordinates. We are adding from the goal to the initial node by using the Map parent. And finally we invert the array just built.



```
private static boolean isGoal(Coordenada current, Coordenada goal) {

    return (current.getX() == goal.getX() && current.getY() == goal.getY());
}

private static Coordenada[] reconstructPath(Map<Coordenada, Coordenada> parent, Coordenada actual, Coordenada goal, int length) {
    Coordenada[] sol = new Coordenada[length+1];
    int i=0;

    Coordenada current = goal;
    while (current.getX() != actual.getX() || current.getY() != actual.getY()) {
        sol[i] = current;
        current = parent.get(current);

        i++;
    }
    sol[i] = current;

    return invertir(sol);
}

private static Coordenada[] invertir(Coordenada[] sol) {
    Coordenada[] res = new Coordenada[sol.length];

    for (int i=0; i<sol.length; i++) {
        res[i] = sol[sol.length-i-1];
    }

    return res;
}
```

Function to invert an array

EXPERIMENTAL RESULTS, PERFORMANCE SLOTS, COMENTS



In the picture we can see a maze where each blank cell is a valid position. Meanwhile, each asterisk represents an obstacle, which means that our algorithm can't use that cell to continue and has to choose another way. As it is shown, the algorithm starts on I (Initial) and looks for a path (represented with the symbol "+") to get to G (Goal) avoiding the obstacles.

Always looking for the most optimistic path