# A* ALGORITHM PROJECT

Lucía Corpas
Javier Jordán
Illya Rozumovskyy
Jorge Velázquez

# INTRODUCTION

In this project, we have developed an agent that is able to find its way in a maze full of obstacles. We have created the maze at random, applied the A* algorithm to find an optimal path, and printed the results to the console.

The maze is a rectangular matrix of 60 rows and 80 columns and each element of the matrix is a possible state of the agent.

A certain fraction of the matrix elements will contain an obstacle that will be chosen at random. The initial and goal states for the agent will also be chosen at random. Therefore, for each execution of the Java application, they should be different.

We have to check that neither the initial nor the goal states are occupied matrix elements. In case that the initial or the goal states are occupied by obstacles, the program must print a message explaining the problem, and halt.

# A* ALGORITHM

The A* algorithm is defined as following:

- Problem formulation:
  - States: a physical configuration.
  - Initial state.
  - State transitions: steps ➡ neighbor nodes
  - Cost of each step: a positive number.
- Solution:
  - Transition sequence that leads from the initial state to a goal state.
  - **Optimal** when has the lowest path cost among all solutions

## A* ALGORITHM

The agent must use the A* algorithm with a suitable heuristic function in order to find a sequence of actions that takes it from the initial state to the goal state, then it must move according to the sequence to reach the goal.

We have implemented the A* algorithm and designed our solution. The data structures that we have used are:

- Sets.
- Lists.
- Normal Arrays.

Because we think they are the most appropriated for this project.

## DEVELOPING THE ALGORITHM

In the project when obstacles do not occupy the initial and the goal states, the output of the program must be a text of 60 rows and 80 columns (4800 text characters total). Each character must be:

- An asterisk (*) if the state is an obstacle.
- A capital I letter (I) for the initial state.
- A capital G letter (G) for the goal state.
- A plus sign (+) if the state belongs to the optimal path found by the A* algorithm.
- A blank space, in all other cases.

If the goal state is not reachable from the initial state, then the program must output the 4800 text characters without printing any optimal path, and after that, it must output a message explaining the problem.

## PROJECT STEPS

01

### MAZE CREATION

Develop the code to generate, manage and print out a maze

02

### A* ALGORITHM

Develop an implementation of the A* algorithm

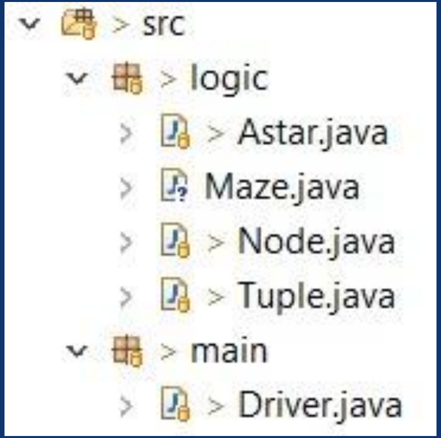
03

### GENERATE PLOTS

Create plots to analyze and discuss the mean length of the optimal path, and the number of times that the goal state is not reachable, as you change the fraction of obstacles in the matrix

# PROGRAM CODE

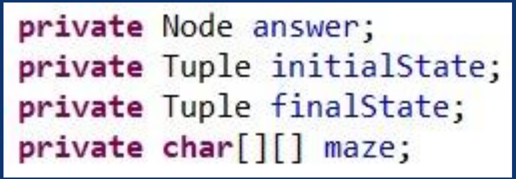
The program is distributed in the following way:



```
src
  logic
    Astar.java
    Maze.java
    Node.java
    Tuple.java
  main
    Driver.java
```

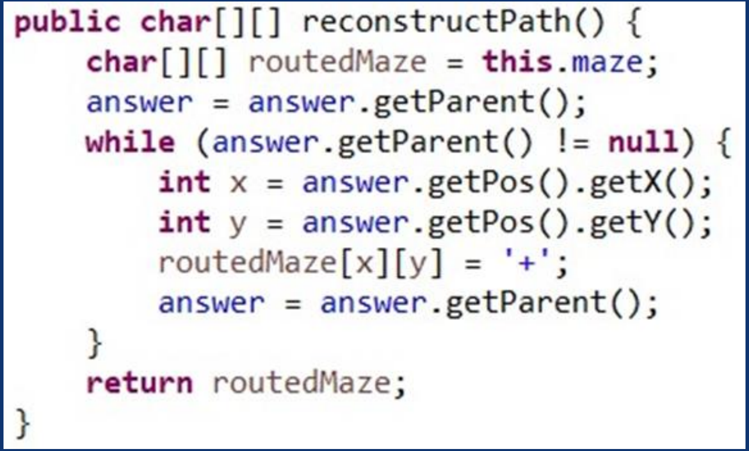
- Package logic:
  - ✓ Astar: A* algorithm
  - ✓ Maze: creation of the maze, inital state, goal state and obstacles.
  - ✓ Node: positions, parent and actual cost (g).
  - ✓ Tuple: java class tuple.
- Package main:
  - ✓ Drive: implements all clases.

## PROGRAM CODE: ASTAR CLASS



```java
private Node answer;
private Tuple initialState;
private Tuple finalState;
private char[][] maze;
```

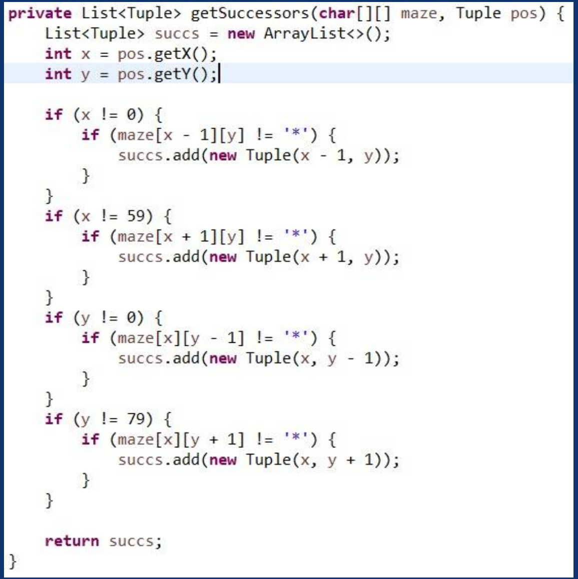
Attributes of the class.



```java
public char[][] reconstructPath() {
    char[][] routedMaze = this.maze;
    answer = answer.getParent();
    while (answer.getParent() != null) {
        int x = answer.getPos().getX();
        int y = answer.getPos().getY();
        routedMaze[x][y] = '+';
        answer = answer.getParent();
    }
    return routedMaze;
}
```

Method reconstructPath() that receives the maze and calculates the optimal path from the goal state to the initial state by means of their parents

## PROGAM CODE: ASTAR CLASS



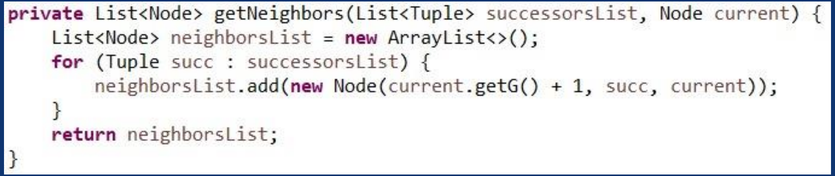
```java
private List<Tuple> getSuccessors(char[][] maze, Tuple pos) {
    List<Tuple> succs = new ArrayList<>();
    int x = pos.getX();
    int y = pos.getY();

    if (x != 0) {
        if (maze[x - 1][y] != '*') {
            succs.add(new Tuple(x - 1, y));
        }
    }
    if (x != 59) {
        if (maze[x + 1][y] != '*') {
            succs.add(new Tuple(x + 1, y));
        }
    }
    if (y != 0) {
        if (maze[x][y - 1] != '*') {
            succs.add(new Tuple(x, y - 1));
        }
    }
    if (y != 79) {
        if (maze[x][y + 1] != '*') {
            succs.add(new Tuple(x, y + 1));
        }
    }

    return succs;
}
```

Method getSuccessors returns a list of tuples of the successors of a determine position.

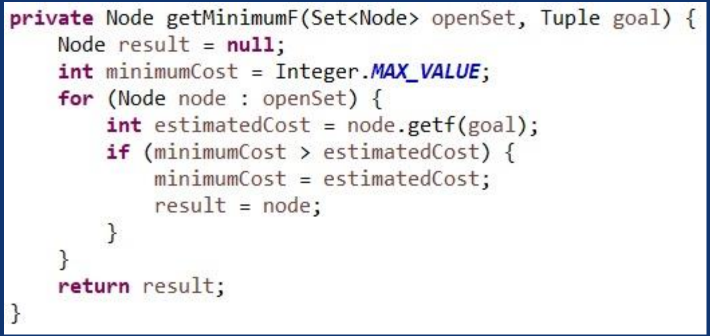
Method getNeighbors returns a list of tuples of the neighbours of a determine position.



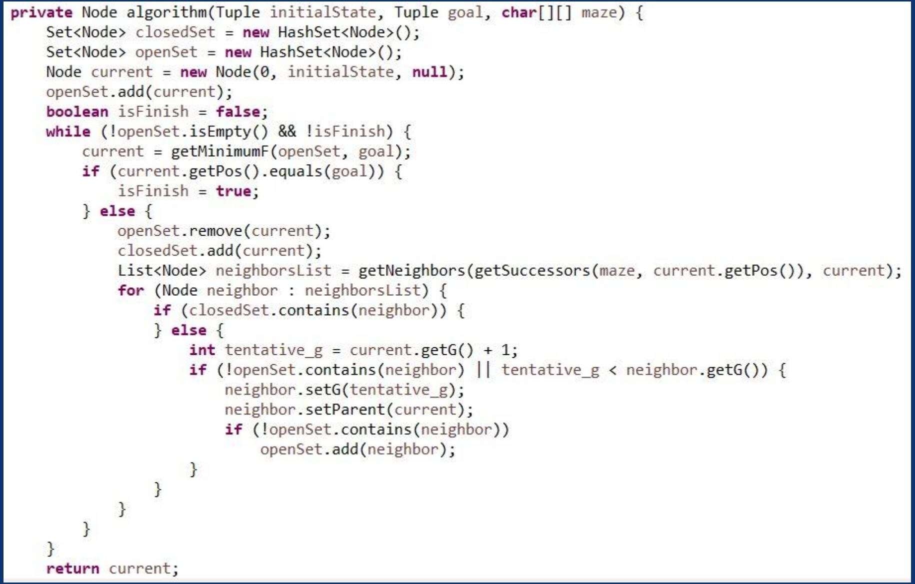
```java
private List<Node> getNeighbors(List<Tuple> successorsList, Node current) {
    List<Node> neighborsList = new ArrayList<>();
    for (Tuple succ : successorsList) {
        neighborsList.add(new Node(current.getG() + 1, succ, current));
    }
    return neighborsList;
}
```

# PROGRAM CODE: ASTAR CLASS

Method getMinimumF returns the node from de openset with the minimum global cost (F).



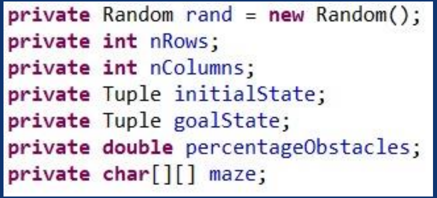
```java
private Node getMinimumF(Set<Node> openSet, Tuple goal) {
    Node result = null;
    int minimumCost = Integer.MAX_VALUE;
    for (Node node : openSet) {
        int estimatedCost = node.getf(goal);
        if (minimumCost > estimatedCost) {
            minimumCost = estimatedCost;
            result = node;
        }
    }
    return result;
}
```



```java
private Node algorithm(Tuple initialState, Tuple goal, char[][] maze) {
    Set<Node> closedSet = new HashSet<Node>();
    Set<Node> openSet = new HashSet<Node>();
    Node current = new Node(0, initialState, null);
    openSet.add(current);
    boolean isFinish = false;
    while (!openSet.isEmpty() && !isFinish) {
        current = getMinimumF(openSet, goal);
        if (current.getPos().equals(goal)) {
            isFinish = true;
        } else {
            openSet.remove(current);
            closedSet.add(current);
            List<Node> neighborsList = getNeighbors(getSuccessors(maze, current.getPos()), current);
            for (Node neighbor : neighborsList) {
                if (closedSet.contains(neighbor)) {
                } else {
                    int tentative_g = current.getG() + 1;
                    if (!openSet.contains(neighbor) || tentative_g < neighbor.getG()) {
                        neighbor.setG(tentative_g);
                        neighbor.setParent(current);
                        if (!openSet.contains(neighbor))
                            openSet.add(neighbor);
                    }
                }
            }
        }
    }
    return current;
```

Method Node algorithm, is the proper A* algorithm which implements the rest of the methods of the class.

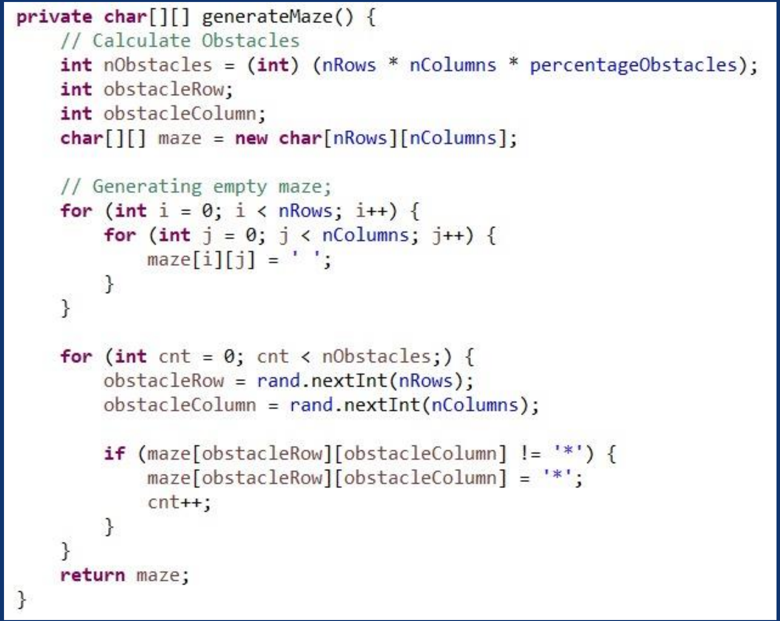
# PROGRAM CODE: MAZE CLASS



```java
private Random rand = new Random();
private int nRows;
private int nColumns;
private Tuple initialState;
private Tuple goalState;
private double percentageObstacles;
private char[][] maze;
```

Attributes of the class

Method generateMaze() returns a maze with its obstacles.



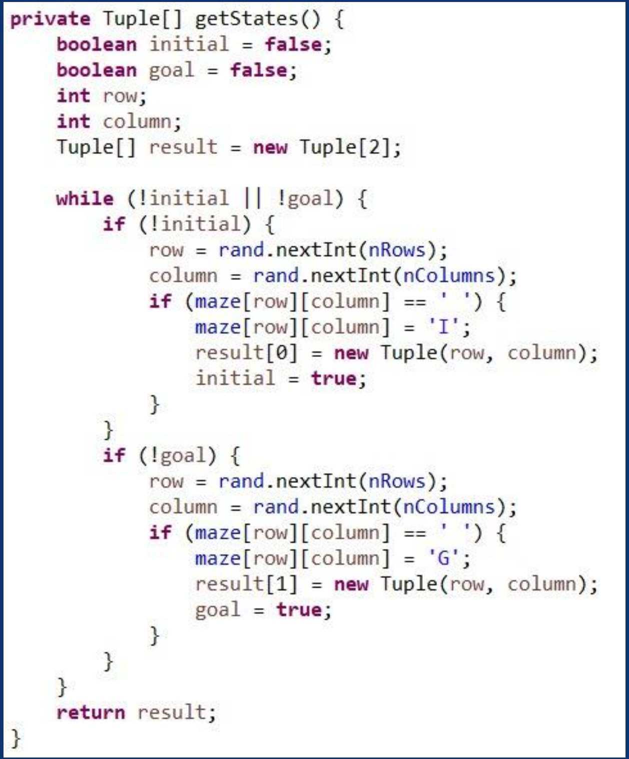
```java
private char[][] generateMaze() {
    // Calculate Obstacles
    int nObstacles = (int) (nRows * nColumns * percentageObstacles);
    int obstacleRow;
    int obstacleColumn;
    char[][] maze = new char[nRows][nColumns];

    // Generating empty maze;
    for (int i = 0; i < nRows; i++) {
        for (int j = 0; j < nColumns; j++) {
            maze[i][j] = ' ';
        }
    }

    for (int cnt = 0; cnt < nObstacles;) {
        obstacleRow = rand.nextInt(nRows);
        obstacleColumn = rand.nextInt(nColumns);

        if (maze[obstacleRow][obstacleColumn] != '*') {
            maze[obstacleRow][obstacleColumn] = '*';
            cnt++;
        }
    }
    return maze;
}
```

# PROGRAM CODE: MAZE CLASS

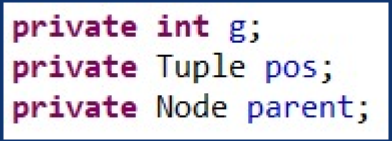


```
private Tuple[] getStates() {
    boolean initial = false;
    boolean goal = false;
    int row;
    int column;
    Tuple[] result = new Tuple[2];

    while (!initial || !goal) {
        if (!initial) {
            row = rand.nextInt(nRows);
            column = rand.nextInt(nColumns);
            if (maze[row][column] == ' ') {
                maze[row][column] = 'I';
                result[0] = new Tuple(row, column);
                initial = true;
            }
        }
        if (!goal) {
            row = rand.nextInt(nRows);
            column = rand.nextInt(nColumns);
            if (maze[row][column] == ' ') {
                maze[row][column] = 'G';
                result[1] = new Tuple(row, column);
                goal = true;
            }
        }
    }
    return result;
}
```

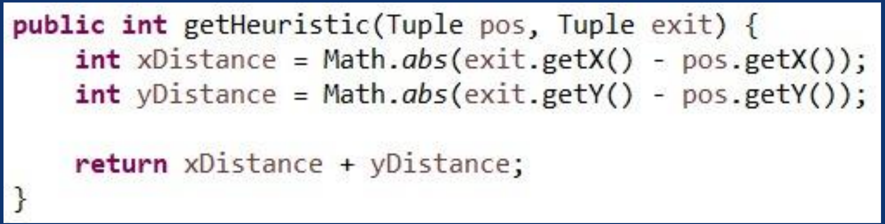
Method getStates() calculates and returns the goal state.

# PROGRAM CODE: NODE CLASS



```java
private int g;
private Tuple pos;
private Node parent;
```

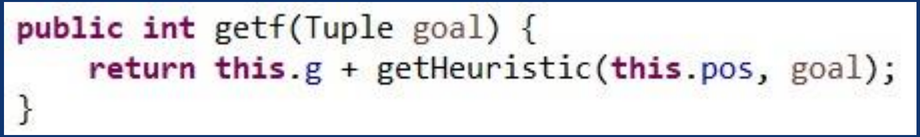
Attributes of the class



```java
public int getHeuristic(Tuple pos, Tuple exit) {
    int xDistance = Math.abs(exit.getX() - pos.getX());
    int yDistance = Math.abs(exit.getY() - pos.getY());

    return xDistance + yDistance;
}
```

Method getHeuristic returns heuristic cost from the current position to the goal state



```java
public int getf(Tuple goal) {
    return this.g + getHeuristic(this.pos, goal);
}
```

Method getf returns the global cost by means of adding the heuristic and actual cost.

## PLOTS

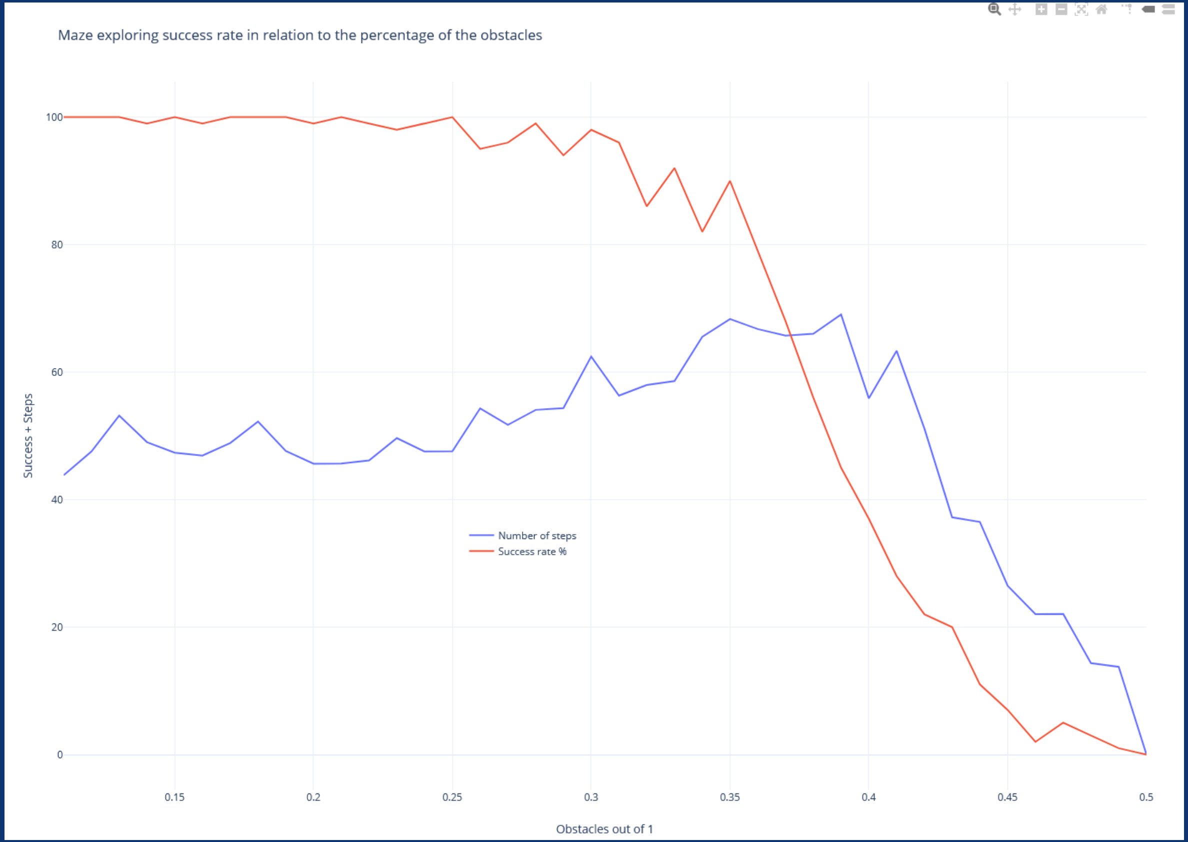
Create plots to analyze and discuss the mean length of the optimal path, and the number of times that the goal state is not reachable, as you change the fraction of obstacles in the matrix.

We used a plot with two variables ‘x’ and ‘y’, where ‘x’ has one entry and ‘y’ has two entries. The entry for ‘x’ is the fraction of obstacles in the matrix, whereas the entries for ‘y’ are the percentage of success and the number of steps to reach the goal state.

We start with 10% of obstacles and increase this value up to 50%. We stop in 50% because at this value the percentage of the A* algorithm success is very low. In this case, initial state and goal state are unreachable.

For each value of ‘x’ we will iterate 100 times and calculate their average, and that will be the value appearing in the plot.

The success' rate breakpoint is when we have a 35% of obstacles. The number of steps decrease as you can only reach the goal state when initial and goal states are near each other.


Maze exploring success rate in relation to the percentage of the obstacles
Success + Steps
Obstacles out of 1
Number of steps
Success rate %