Resolution of Labyrinth Robots using Search Methods in the Language C++ (Theme 7/Group 26)

Beatriz Mendes
Mestrado Integrado Eng. Informática
Faculdade de Engenharia da UP
Porto, Portugal
up201604253@fe.up.pt

Fernando Alves
Mestrado Integrado Eng. Informática
Faculdade de Engenharia da UP
Porto, Portugal
up201605270@fe.up.pt

Henrique Gonçalves
Mestrado Integrado Eng. Informática
Faculdade de Engenharia da UP
Porto, Portugal
up201608320@fe.up.pt

***Abstract*—In this article, it will be featured a searching problem in the form of a puzzle. The main goal is to not only implement it and solve it using several searching methods but also analyze it and reach a conclusion on which algorithm represents the optimal approach. We will present a formulation of the problem, which discloses the representation of the game state; the initial state: the test objective; the operands involved.**

I. Introduction

This article was proposed in the scope of the third year subject Artificial Intelligent, taught at Faculdade de Engenharia da Universidade do Porto.

The main goal of this project is to implement a single-player puzzle and solve multiple levels with different searching methods. The purpose of this is to study these different methods in search for the optimal solution, that is, the less cost-worthy.

We chose the game ***Labyrinth Robots*** to embody the searching problem proposed.

The rest of this article will respect the following structure:

1) Description of the Problem
2) Formulation of the Problem
3) Related Work
4) Implementation of the game
5) Search Algorithms
6) Experiences and Results
7) Conclusion and Perspectives of Development

II. Description of the Problem

As stated before, ***Labyrinth Robots*** is a single-player puzzle. There are two types of characters: the players and the "helpers". The goal is to move the playing robots through the mazes in order to reach their targets (see Images 1 and 2). A "helper" robot's single purpose is to help the playing robots reach their targets, therefore not having any target of their own.

When a robot starts moving in a single direction, it keeps moving until it reaches an obstacle or another robot.

The maze maps have multiple valid solutions. However, it is intended to find the less cost-worthy path, that is, the shortest path possible in terms of moves made.

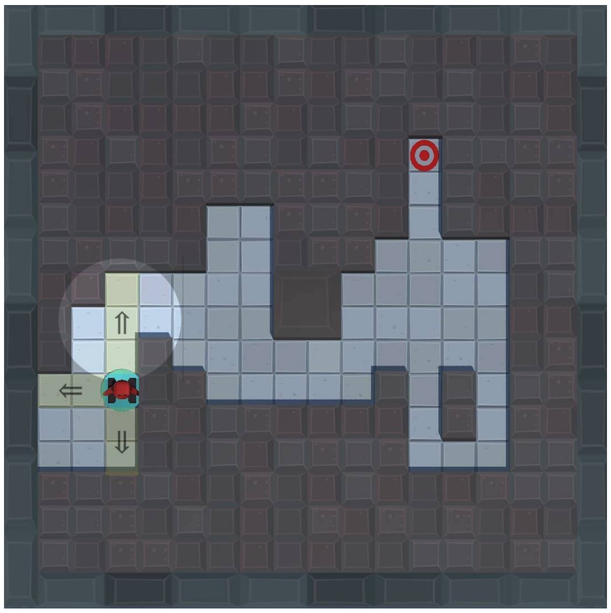


Fig. 1. Possible moves

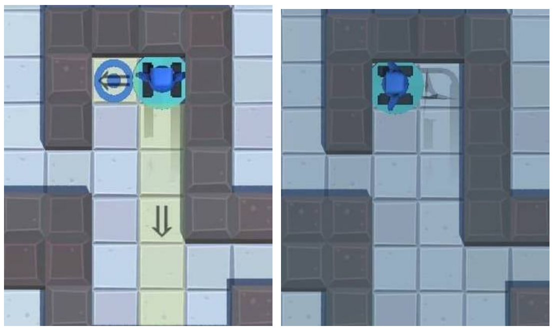


Fig. 2. Winning move

III. Formulation of the Problem

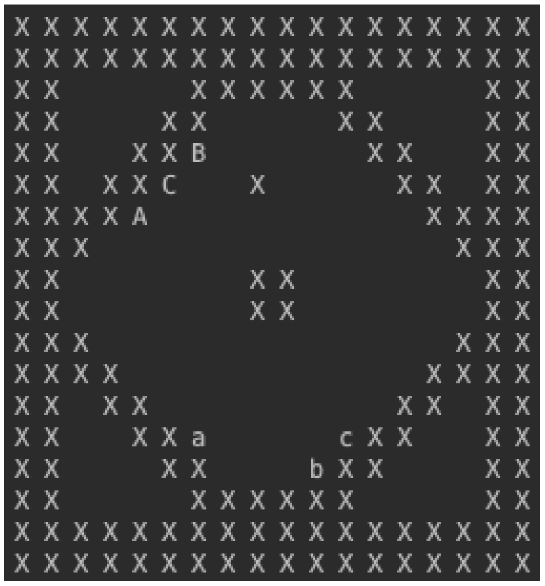
A. Representation of the Game State

The game state is represented by the Map class containing the maze layout and the list of robots present in the level. Seeing as there are various levels, there is a list of maps in which each map represents a single level.

A layout is represented by a bi-dimensional matrix of *chars*, in which 'X' represents a wall; ' ' represents an empty cell; other symbols (e.g.: 'A') represent a robot's target cell.

Each Robot stores its own current coordinates, type (Helper or Playing Robot) and corresponding char: note that the Robot's char is always the lowercase of its target cell's char (e.g.: 'a'-robot; 'A'-target cell).

A map is immutable, that is, its values are unalterable. This makes it so a wall or target cells cannot be moved.



```
X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X
X X         X X X X X X         X X
X X       X X         X X       X X
X X     X X B           X X     X X
X X   X X C     X         X X   X X
X X X X A                   X X X X
X X X                         X X X
X X             X X             X X
X X             X X             X X
X X X                         X X X
X X X X                     X X X X
X X   X X                 X X   X X
X X     X X a         c X X     X X
X X       X X       b X X       X X
X X         X X X X X X         X X
X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X
```

Fig. 3. Representation of a level

B. Initial state

The initial state in relation to the current level. Each level has a particular number of robots in play; this exact number of robots are positioned in their initial coordinates, which are also stored in the map.

C. Objective Test

The puzzle is solved when all playing robots are in their respective target cells.

D. Operands

The variables *"line"* and *"col"* symbolize the line and column that a robot occupies in the map. The superior left corner maps the cell with the coordinates (0,0) and thenceforth the values grow from left to right and from top to bottom.

For each robot, there are the following operands available:

Name: moveUp
Prerequisites: (line-1,col) != 'X' and (line-1,col) != Robot
Outcome: line decrements until an obstacle is found
Cost: 1

Name: moveDown
Prerequisites: (line+1,col) != 'X' and (line+1,col) != Robot
Outcome: line increments until an obstacle is found
Cost: 1

Name: moveLeft
Prerequisites: (line,col-1) != 'X' and (line,col-1) != Robot
Outcome: col decrements until an obstacle is found
Cost: 1

Name: moveRight
Prerequisites:(line,col+1) != 'X' and (line,col+1) != Robot
Outcome: col increments until an obstacle is found
Cost: 1

IV. Related Work

In preparation for this project, we researched preexisting solutions and found a few implementations that are in some way similar to our problem. Of these, some should be highlighted such as :

1) *"Jeff Chiu - Graph Maze Problem"*[1] proposes a solution to a maze very similar to ours in Python. Chiu, however, simply explores pure Breadth and Deep First Search algorithms.
2) *"Maze Search"*[2] is developed in Java. It explores a few more algorithms such as Greedy and A* search.
3) *"Ricochet Robot - Implementation of a board game called Ricochet Robot including a GUI and a very fast solver"*[3] is an implementation of a solution which was initially developed in Python and was later optimized in C. This solution not only explores Breadth First Search and Iterative Deepening but also implements some improvements to the original algorithms.

While we looked over all solutions, we decided to inspect more closely the Ricochet Robot's seeing as it has an extra improvement level but also due to the fact that it's available in C, which is semantically more identical to C++.

V. Implementation of the game

As the program starts, the user can choose to solve the game himself or see how our programmed bot solve it. The user is then presented with a few other menu choices such as the search method and the characteristics associated (if applicable) and the level.

Furthermore an instance of the class Game is created. This class is responsible for reading from a text file - *'readDataFromFiles'* - the information of all available levels along as initiating the game cycle correspondent to the game mode previsouly chosen - *'soloMode()'* or *'botMode()'*.

A. Representation of the Game State

As previously stated, a level/map contains the maze which is stored in a vector of char vectors and the robots which are stored in a vector of objects of the class Robot. It also represents the current state of the game. The level is displayed through the function *'displayWithRobots()'*, demonstrated in figure 3.

B. Human mode

The user is asked where and what robot they wish to move. The input request is two consecutive chars. The first char represents the robot and the second char represents the direction: 'a' - left; 'd' - right; 'w' - up; 's' - down. For example, the move "aw" means the user wished to move the robot 'a' upwards.

In the game cycle, the input is checked and interpreted - '*interpretInputDir()*' and '*interpretInputRobot()*'.

If the prerequisites for these operands are met, the function *'moveRobot()'* will move the robot in the stated direction until an obstacle is found.

This mode also offers the possibility to ask for hints by writing 'hh'. For this feature, we chose to use A* search due to its better overall efficiency, as we will conclude later on.

C. Bot mode

On the assumption that the map is solvable, the game cycle displays and executes iteratively the sequence of moves calculated by the search method - *'moveRobot()'*. If the method is unable to reach a solution, it will simply print out "Algorithm could not find an answer".

In both cases, the relative performance statistics are displayed: **steps** (if map is solvable) , **expanded nodes** and **elapsed time**.

VI. Search Algorithms

In the development of this article, we considered six searching methods, such as: Depth First Search; Breadth First Search; Iterative Deepening Search; Uniform Cost Search; Greedy Search; A* Search.

Of these six, we can expect to divide them into algorithms which focus on finding an optimal solution and algorithms focus more on reaching ***a*** solution - the first one they reach.

The focus point of the algorithms are the nodes. In this case, a node represents a possible game state. To represent them we created a *struct Node* (see Listing 1).

```
typedef struct{
    vector<Robot> robots;
    int depth = 0;
    int heuristic = 0;
    int expansions = 0;
    vector<pair<int, int>> moveSeq = {};
} Node;
```

Listing 1. Node Data Structure

All algorithms have a starting node and the current state (map layout and robots) as a starting point. Furthermore, all have common data structures such as: a ***priority queue*** that stores the nodes to be expanded and the corresponding compare function - '*p_queue*'; a ***map*** of the previous analysed states - '*prevStates*'. The A* and Greedy algorithms use heuristics to choose the next node to expand. For this project we developed 5 heuristics. However, the first one simply returns 0, so we will not mention it much; The second heuristic checks if a playing robot is aligned with its final position. If not, the value of the heuristic rises; The third heuristic is an improvement on the previous one. In the case that the playing robot is aligned with its final position, we check if that path has obstacles (other robots or walls). If it does, then the heuristic rises; The fourth heuristic checks the density of obstacles in the area between the robot and its target, and increases the heuristic accordingly; The fifth and final heuristic is our best one overall. It takes the principle from the heuristic nº3 and also checks if the robot has a way to stop in the final position (in order to stop, the robot has to collide with something). If he does not, the heuristic increases.

A. Depth First Search

In this algorithm, the priority queue prioritizes nodes with greater depth.

Algorithm 1 DFS Algorithm

```
1: function DFS(Node startNode,Map currentMap)
2:     p_queue.push(startNode)
3:     while !queue.empty do
4:         node ← p_queue.top
5:         p_queue.pop
6:         if current_depth < MAX_DEPTH then
7:             continue                          ▷ Prevents cycles
8:         for each robot do
9:             for each direction do
10:                n_expansions++
11:                if robot move is useful then
12:                    get new node
13:                    if game is over then
14:                        return node
15:                    else
16:                        save current state
17:                        if node isn't in prevStates then
18:                            if visited w/ > depth then
19:                                p_queue.push(node)      ▷ Save w/ < depth possible
20:                        else
21:                            continue
22:                    else
23:                        push to p_queue
24:    node.depth = -1 ▷ Means no solution was found
25:    return node
```

In a nutshell, the algorithm starts with the node which represents the robot's initial position and keeps expands the node leftmost. It will store a nodes if: it hasn't been visited before; it has been visited but with greater depth; if the move is useful (e.g.: moves directly towards a wall, this move is useless) (see Algorithm 1).

B. Breadth First Search

In the Breadth First Search, the queue prioritizes nodes with least depth.

Algorithm 2 BFS/Uniform Cost Algorithm

```
 1: function BFS(Node startNode,Map currentMap)
        (...) //Equal to Algorithm 1
 2:     if current_depth < MAX_DEPTH then
 3:         continue            ▷ Stops infinite searching
        (...) //Equal to Algorithm 1
 4:     if game is over then
 5:         return node
 6:     else
 7:         save current state
 8:         if node isn't in prevStates then
 9:             continue
10:         else
11:             p_queue.push(node)
          (...) //Equal to Algorithm 1
12:     node.depth = -1 ▷ Means no solution was found
13:     return node
```

The implementation of this search method is quite similar to the previously stated method, Depth First Search. Besides the priority queue comparison, the other difference in the implementation is the verification between the lines 18 - 21 of the BFS algorithm: this search already expands the nodes with lesser depth, so these statements would be redundant (see Algorithm 2).

C. Uniform Cost Search

Seeing as, in this specific problem, all nodes have an unitary cost - cost(n)=Depth(n) - this algorithm is equivalent to the Breadth First Search (see Algorithm 2).

D. Iterative Deepening Search

As the name of the method suggests, this algorithm is a take on the Deep First Search already implemented.

This algorithm defines a temporary maximum depth - '*tempMaxDepth*', starting at 0. When all nodes with maximum depth '*tempMaxDepth*' have been visited: the '*prevStates*' map which stores all previous states is cleared; the '*tempMaxDepth*' is incremented; another iteration begins (see Algorithm 3).

Within each iteration, the nodes are expanded using a Deep First Search.

Algorithm 3 IDFS Algorithm

```
 1: function IDFS(Node startNode,Map currentMap)
 2:     tempMaxDepth ← 0
 3:     p_queue.push(startNode)
 4:     while current_depth < MAX_DEPTH do
 5:         tempMaxDepth++
 6:         n_expansions ← 0
 7:         clear prevStates
 8:         p_queue.push(startNode)
 9:         while !p_queue.empty do
10:             first ← p_queue.top
11:             p_queue.pop
                //Prevents cycles
12:             if first.depth < MAX_DEPTH then
13:                 continue
14:             for each robot do
15:                 for each direction do
                        (...) //Equal to Algorithm 1
16:     node.depth = -1 ▷ Means no solution was found
17:     return node
```

E. Greedy Search

In the Greedy Search, the queue prioritizes nodes with least heuristic points, which estimates how close we are to the solution.

Algorithm 4 Greedy Algorithm

```
 1: function GREEDY(Node startNode,Map currentMap)
 2:     startNode.heuristic ← calcHeuristic()
 3:     p_queue.push(startNode)
 4:     n_expansions ← 0
 5:     while !queue.empty do
 6:         node ← p_queue.top
 7:         p_queue.pop
 8:         if current_depth < MAX_DEPTH then
 9:             continue                ▷ Prevents cycles
10:         for each robot do
11:             for each direction do
12:                 n_expansions++
13:                 if robot move is useful then
14:                     get new node with specific heuristic
                (...) //Equal to Algorithm 1
15:     node.depth = -1 ▷ Means no solution was found
16:     return node
```

The implementation of this search method is quite similar to the previously stated method, Depth First Search. The only differences are that the priority queue comparison is now made in terms of heuristic and not depth; and when a new node is inserted it includes its heuristic (see Algorithm 4).

F. A* Search

In the A* Search, the queue prioritizes nodes with least sum of heuristic points with the current depth.

Algorithm 5 A* Algorithm

```
1: function ASTAR(Node $startNode$,Map $currentMap$)
2:     $startNode.heuristic \leftarrow calcHeuristic()$
3:     $p\_queue$.push($startNode$)
4:     $n\_expansions \leftarrow 0$
5:     $minDepth \leftarrow 0$
6:     while !$queue.empty$ do
7:         $node \leftarrow p\_queue$.top
8:         $p\_queue$.pop
9:         if node.depth > minDepth then
10:            return $bestVal$        ▷ Already found best solution, return
11:        if current_depth < MAX_DEPTH then
12:            continue                ▷ Prevents cycles
13:        for each robot do
14:            for each direction do
15:                $n\_expansions$++
16:                if robot move is useful then
17:                    get new $node$
18:                    if game is over then
19:                        if node.depth < minDepth then
20:                            $minDepth \leftarrow node.depth$
21:                            $bestVal \leftarrow node$        ▷ New Better Solution
                (...) //Equal to Algorithm 1
22:    $node$.depth = -1 ▷ Means no solution was found
23:    return $node$
```

Contrary to the Greedy search, the algorithm only exits after finding the guaranteed best solution possible, that is, the solution with the least amount of required steps.

VII. Experiences and Results

In order to be able to properly analyse the efficiency of the algorithms implemented, we collected information on how they performed under certain circumstances.

We tested them on: multiple levels which varied in size, number of robots and overall difficulty and different heuristics (when). For each of these tests, we recorded the **steps**, **expanded nodes** and **elapsed time**. All the tests were conducted in a virtual machine running 3 of the cores of an AMD Ryzen 5 processor with 4GB of RAM.

We registered all the obtained data in tables, present in Appendix A. After analysing it, we selectively gathered the most relevant information and organized it into graphs. Some of the values of the tables are left with a "-" when the algorithm failed to obtain an answer in a time-frame of 90 seconds.

All presented graphs have a common study variable: the levels. While we implemented a total of 7 levels, we decided to not include the first level due to the fact that it was mostly a testing level with values significantly smaller, making it difficult to represent in the same scale.

A. Heuristics

1. A* Search

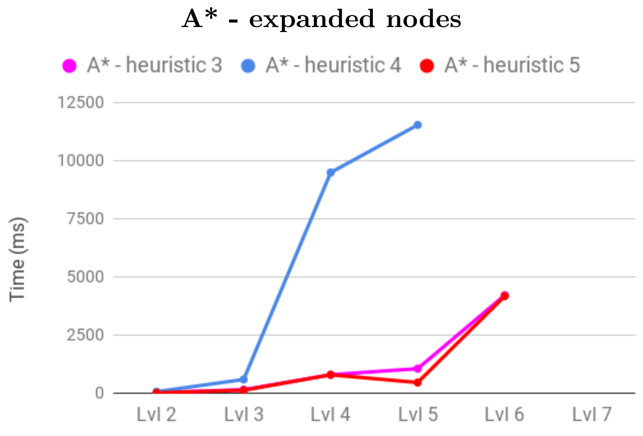


Fig. 4. Variation of A* expanded nodes in relation to levels(2-7)

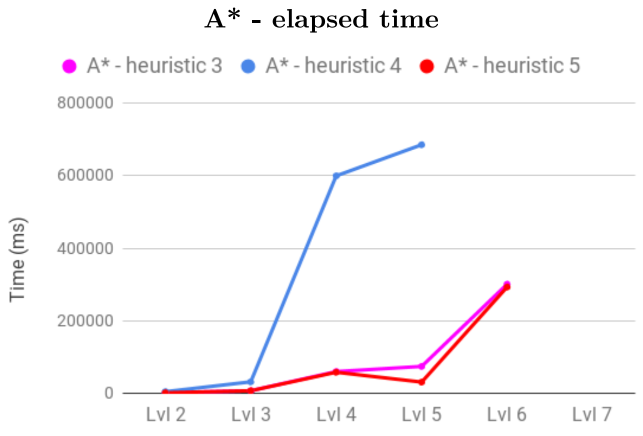


Fig. 5. Variation of A* elapsed time(ms) in relation to levels(2-7)

These two graphs have an almost identical stain. Therefore we can easily conclude that, although the algorithm with the 4th and 5th heuristic are almost tied, the latter surpasses in level 5.

2. Greedy Search

The tables for the Greedy algorithm results had several cases of the different heuristics not being able to reach a solution in the established time-frame, so we decided to not include graphs for the results of this algorithm with different heuristics. We did include the best results of each algorithm in the graphs of tables I, II and III, for comparison.

B. Algorithms

This section analyses all algorithms back to back. Seeing as there are many versions of the A* and Greedy algorithms, depending on the heuristics used, we decided to pick the more efficient one overall. Therefore, we will be taking in consideration A* and Greedy with the 5th heuristic.

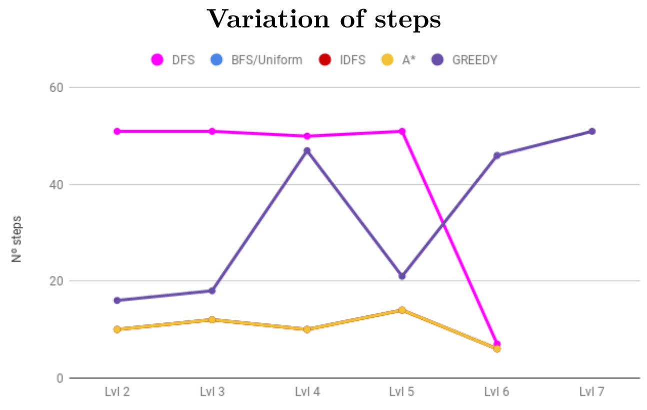


Fig. 6. Variation of algorithms steps in relation to levels(2-7)

In Fig. 6, A*, BFS and IDFS have identical behaviour. The IDFS and BFS line is "hidden" behind the yellow line. Therefore, we can state that BFS, IDFS and A* all tend to reach the optimal solution while the DFS and Greedy focus solely on finding a single solution.

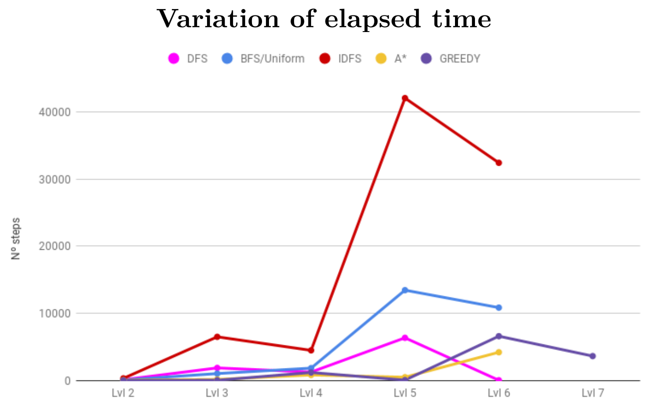


Fig. 7. Variation of algorithms elapsed time(ms) in relation to levels(2-7)

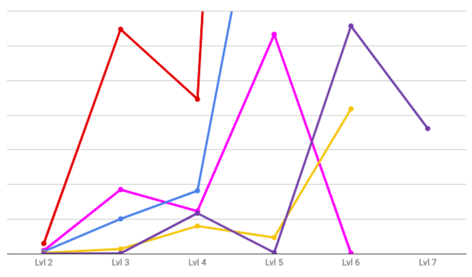


Fig. 8. Close up of previous Fig. 7

In Fig. 7 and 8, A* and Greedy stand out immensely. Their values of elapsed time are so much smaller that are only truly visible when the scale is enlarged. From these graphs, we can establish that A* and Greedy reach a solution, in average, at the same time.

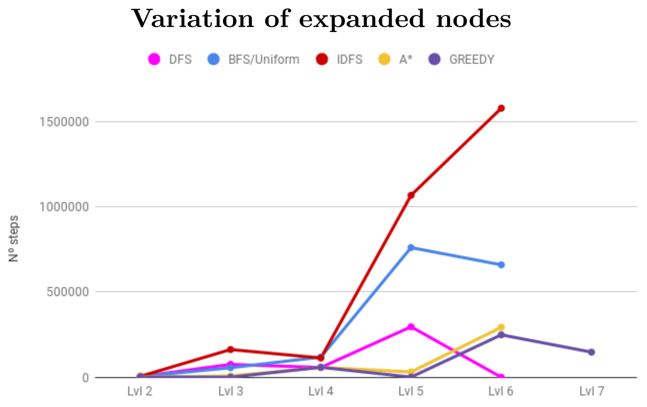


Fig. 9. Variation of algorithms expanded nodes in relation to levels(2-7)

This is possibly the most disperse graph of all. It allows us to determine that the algorithms which expand more and least nodes are IDFS and Greedy respectively.

VIII. Conclusion and Perspectives of Development

In conclusion, we were able to develop all the algorithms required for this project, and were also able to implement several heuristics specific to our puzzle. After running all the algorithms, we were able to conclude that, for our puzzle, the fastest algorithm to obtain an answer is usually the Greedy using the heuristic n°5. However, this algorithm only provides a possible solution, not the ideal solution. If we want the optimal solution, the best option is to use the A*, with heuristic n°5 as well. It guarantees and optimal solution, and sometimes was even faster than the Greedy algorithm, which was surprising.

Overall the results we obtained were fairly close to what we expected. However we were surprised by the fact that the IDFS algorithm expanded more nodes in the iteration that it found a solution than the BFS algorithm did in total. After some investigation, we found that this was due to the fact that we check if a state was already expanded. In the case of the BFS, if we reach a state for the first time, we can guarantee that the path to reach that node was the shortest, so any other path that leads to that path is immediately discarded. In the case of the IDFS, since it is based on the DFS method of search, we cannot make this guarantee.

The last level (level seven) we included in our project is the hardest one we could find, and only the Greedy algorithm was able to solve it. We tried to develop an heuristic (heuristic n°5) that would allow our A* algorithm to solve that level, but we were unsuccessful. However, this new heuristic turned out to be our best heuristic overall, since it took into consideration the fact that robots need something to collide with to stop at a given spot.

The only possible improvements we could think of to this project would be an heuristic that could allow the A* algorithm solve the final level. Besides that, we could have possibly added some measure to make the order of simulated moves more random, so that certain algorithms like the DFS would have been a bit more efficient. Other than that we believe we were able to optimize the rest of our code fairly well.

References

[1] Jeff Chiu, “Graph Maze Problem”, 2013, available at: https://github.com/Jeffchiucp/graph-maze-problem , consulted on March 2019.

[2] “Maze Search”, 2018, available at: https://github.com/emuro2/Maze-Search/blob/master/Algorithms.java, consulted on March 2019.

[3] Michael Fogleman, “Ricochet Robot - Implementation of a board game called Ricochet Robot including a GUI and a very fast solver.”, 2012, [online], available at: https://www.michaelfogleman.com/projects/ricochet-robot/ , consulted on March 2019.

Appendix A
Algorithms analysis

	DFS	BFS/Uniform	IDFS	A*	Greedy
Lvl 1	13	35	15	31	19
Lvl 2	3938	4182	5698	1692	196
Lvl 3	75648	55942	162623	7320	335
Lvl 4	57252	118128	112973	58338	58415
Lvl 5	296037	760360	1067031	31205	940
Lvl 6	121	659127	1577399	293195	248865
Lvl 7	-	-	-	-	147305

Table I. Number of nodes the algorithm expanded per level

	DFS	BFS/Uniform	IDFS	A*	Greedy
Lvl 1	4	3	3	3	4
Lvl 2	51	10	10	10	16
Lvl 3	51	12	12	12	18
Lvl 4	50	10	10	10	47
Lvl 5	51	14	14	14	21
Lvl 6	7	6	6	6	46
Lvl 7	-	-	-	-	51

Table II. Number of steps the solution provided by the algorithm has per level

	DFS	BFS/Uniform	IDFS	A*	Greedy
Lvl 1	0,2	0,6	0,5	0,5	0,3
Lvl 2	89,5	62,1	294	25	3,2
Lvl 3	1850	1005	6483	133	8
Lvl 4	1228	1816	4466	798	1167
Lvl 5	6335	13438	42068	466	29
Lvl 6	8,8	10835	32419	4182	6579
Lvl 7	-	-	-	-	3613

Table III. Time taken for algorithms to reach a solution per level

A. *A* heuristics analysis*

	A* - h2	A* - h3	A* - h4	A* - h5
Lvl 1	31	31	51	31
Lvl 2	2513	1740	4761	1692
Lvl 3	8060	7332	31902	7320
Lvl 4	75760	60229	599652	58338
Lvl 5	75383	74413	685472	31205
Lvl 6	387239	301103	-	293195
Lvl 7	-	-	-	-

Table IV. Nodes A* expanded to reach a solution per level with each heuristic

	A* - h2	A* - h3	A* - h4	A* - h5
Lvl 1	0,6	0,5	0,9	0,5
Lvl 2	37	26	72	25
Lvl 3	150	156	597	133
Lvl 4	1069	801	9499	798
Lvl 5	1080	1063	11547	466
Lvl 6	4409	4222	-	4182
Lvl 7	-	-	-	-

Table V. Time taken for A* to reach a solution per level with each heuristic

B. *Greedy heuristics analysis*

	Greedy - h2	Greedy - h3	Greedy - h4	Greedy - h5
Lvl 1	19	19	19	19
Lvl 2	194	225	1574	196
Lvl 3	338	335	519	335
Lvl 4	-	-	-	58415
Lvl 5	5497	5497	146437	940
Lvl 6	-	-	-	248865
Lvl 7	121791	219251	-	147305

Table VI. Nodes Greedy expanded to reach a solution per level with each heuristic

	Greedy - h2	Greedy - h3	Greedy - h4	Greedy - h5
Lvl 1	0,6	0,3	0,3	0,3
Lvl 2	3,5	3,8	29	3,2
Lvl 3	8	8,3	12,9	8
Lvl 4	-	-	-	1167
Lvl 5	115	115	3473	29
Lvl 6	-	-	-	6579
Lvl 7	2928	5886	-	3613

Table VII. Time taken for Greedy to reach a solution per level with each heuristic