Capstone Project

Thiago Lobato

Machine Learning Engineer Nanodegree

December,15th,2016

Plot and Navigate a Virtual Maze

- Definition

1.1 Project overview

This project was based in the Micromouse competition, and the goal is to guide a robot in a maze from a start position to the goal (the center of the maze). The robot is allowed to run the maze twice, the first one freely to just map the maze and the second one to go to the center as fast as possible. This project handles the programming of such a robot with mapping and shortest path skills in a deterministic environment, where the robot has a orientation and receives sensor data of wall distances as it walks through the maze, the robot is allowed to move in only four directions, up, left, down and right.

1.2 Problem Statement

The problems in this project could be divided in three areas, mapping, exploring and obtain the fastest path. The goal is to make a robot that can identify its location and orientation, handle and map the maze walls/obstacles, explore the maze and, finally, be able to find the best path to the goal.

The solution approach will be to map the robot position based in its past position plus actual movement and, at the same time, map the maze walls with the sensor data, to start to build an image of the problem while exploring.

The exploration should be enough to find, if not the best, at least a good path to the goal without too many exploration steps. The way that this will be approached is trying to explore key points in the maze that would allow the discovery of a good path without using too much time in exploration.

The best way will then be achieved with a best path algorithm, which will use the already explored maze data to find the best direction in function of the actual robot position.

1.3 Metrics

The metric of this project is a combination of the first and second run of the robot. It is defined as the number of steps in the second run plus 1/30 of the number in the first one and it is automatic calculated in the given tester.py script. This metric is good to represent real Micromouse competitions, where the first run has a lower weight than the last, therefore it makes sense to try to explore more in the first run.

• Analysis

2.1 Data Exploration

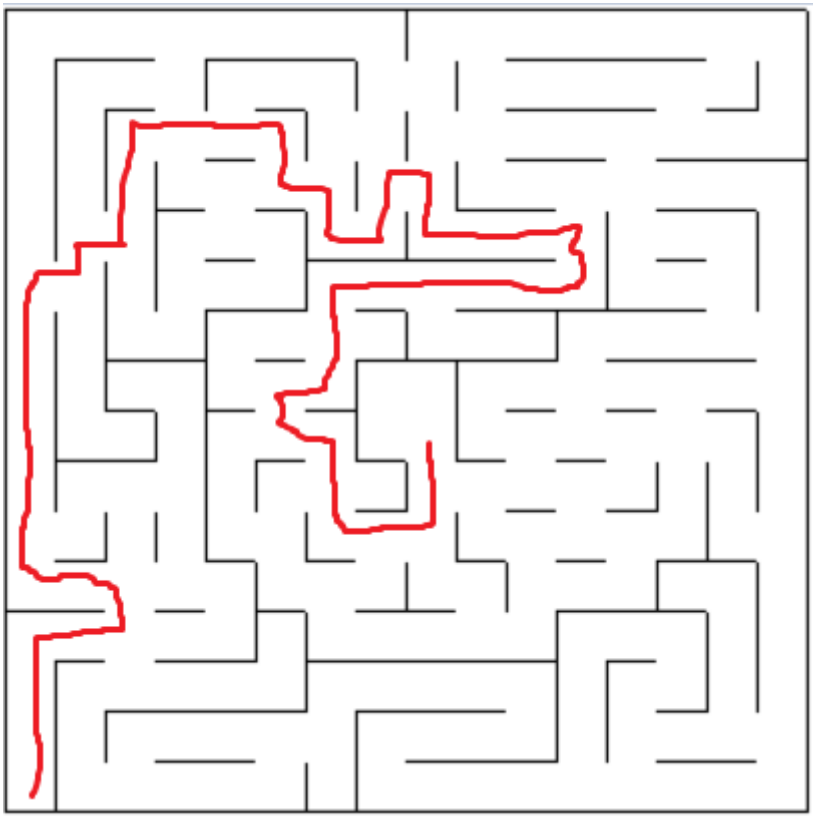
The maze in which the robot will be in is considered a discrete grid maze. The robot itself is considered to always be in the center of each grid and has some orientation ('up', 'down', 'left', 'right') as well as some position, which correspond to his actual grid position in X and Y coordinates, implemented as a [X,Y] list. He has sensors for which it receives a list of 3 values, each being the actual distance from its actual position to a wall on its left, front and right respectively (all based in its current orientation). Only its actual position is stored, and the walls are stored in obstacle matrixes as soon as they are seen, this will be later more detailed discussed. The robot moves in time steps, and in each one he is allowed to change his orientation in +/- 90 degree and to move a number of steps in one direction (X or Y) ranging from -3 (opposite direction) to 3, inclusive. If it finds a wall, it will stop and will not go through.

One of the three mazes provided (most specific the number 3) has many ways of going to the center. It is closed in most direct path to the center and needs one to go around up to the end of the maze to turn back and find the solution, it has many redundant paths, as example in the top left and top right which can cost many time steps. The solutions differ a lot in length, range from 49 squares going in the right way from

up, right, down and avoiding useless obstacles going to more than 60 if ones chooses to use redundant ways.

2.2 Exploratory Visualization

The maze described before can be seen below with its best path:



2.3 Algorithms and Techniques

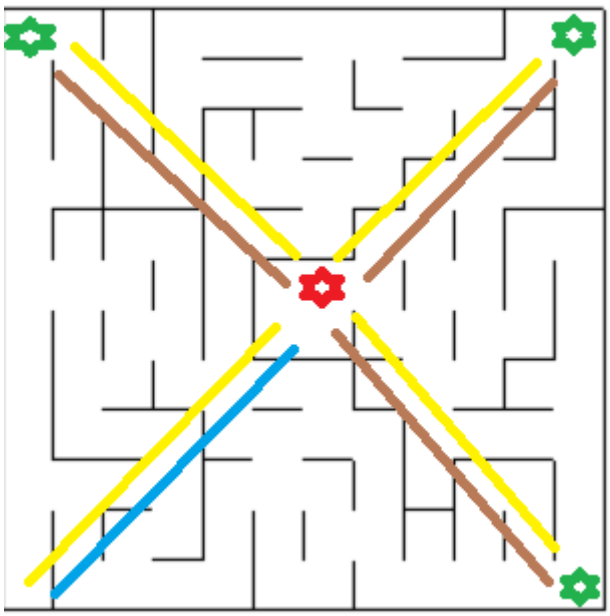
In this problem, all the sensor data is deterministic, which means we don't have to deal with uncertainty. Therefore one can use a approach that maps the position and orientation of the robot based in its previous position and movement. As for example, if the actual position is [2,4] and orientation 'up', after a 2 steps to the right movement and change of orientation of 90 degree clockwise, the new actual position would be [4,4] with orientation 'right'. Only the actual position needs to be tracked.

In the mapping, it was important to identify the wall positions (which would prevent movement) to clearly represent the maze. That was accomplish dividing the walls in: barriers from up ,left ,down and

right. So each one would have its own matrix of mostly 0's where the 1's would indicate a wall in that position, which means that, in that square, the robot is not allowed to go in the specific direction.

The exploring was done as following: the agent would do a 'swing' exploring, going to each corner (key points) and then back to the center, and, in the end, would go back to the start, in that way the agent is able to get a great big picture of the maze and find mostly of the ways to the center (and hopefully also the best) with an efficiently exploring.

The way the robot moves to each key point is with A*, where it tries to reach the goal and change its path as soon as obstacles appears, doing the exploration in the maze while going to key points and later, in the second run, uses the discovered map to find the fastest way. Since the A*, in the beginning, sees a completely obstacle-free path and discovers it along the way, one can use this to try to find the path between the center and each key point, mapping possible solutions to the center. In the figure below one can see how it would work. Yellow lines are going to the center and brown from the center, the blue one is the end where it goes back to the start. This is obviously not the path, but the relation Position x Goal. The red point is the goal and green ones example key points, the example is in maze 1.



The A* algorithm is a graph search tool, that works with nodes and edges, it explores the maze prioritizing the way in which it is more likely to be the best one (closer to the goal), this is done using a heuristic function that, in this case, is the distance from the actual location to the goal position, each square from the grid has its own heuristic function. From the actual position a grid expansion in all possible directions begin, and the point where the next expansion will be is the one closest to the actual location in number of steps (a breath first search), A*, however, using that heuristic function, puts an extra cost in the order of expansion, allowing the algorithm to expand first in the direction closer to the goal.

2.4 Benchmark

The most obvious benchmark of that problem would be an agent that would make just random moves. That agent would inconsistently reach the goal and with an undefined number of steps, a statistic result of 6 random tries gives the following result for the first maze:

	Steps train	Steps real	Score
1	1000	-	-
2	317	683	-
3	1000	-	-
4	1000	-	-
5	897	103	-
6	361	572	584.067

If not even in the simplest maze a random robot can have a good result it is clear that a random agent would not be able to get to the goal with consistency and even more have a good result.

Another more reasonable benchmark would be a count expansion, where the agent tries to reach the goal in a breath first type of expansion, opening every square around it until find the goal and then do a reverse way to go back to the start grid from the destination, building its way through the maze. The results for all mazes can be seen below for the case where the agent reaches the goal and then start the next run:

	Steps train	Steps real	Score
Maze 1	32	43	44.1
Maze 2	55	47	48.867
Maze 3	74	48	50.5

The count method gives a most consistent result and can be used as a benchmark since is the easiest and most intuitive method that works consistent .

Methodology

3.1 Data Preprocessing

In this project, all the data was deterministic and provided through the model with the sensor data, location and mazes environment. Therefore no data preprocessing was needed.

As commented before, the sensor data is a list of 3 values that gives the distances from walls in the left, top and right from the robots orientation in its actual position. This is used to store the obstacle location in matrixes that are later use in the A* algorithm to path optimization.

3.2 Implementation

The robot sensor data was used to create the obstacles matrix with a mapping function for indexing, then, using all those matrixes, a A* algorithm function would give as output the actual movement in direction to the goal, that then is mapped to the robot orientation and position to give the right output. That goal change between given key points as soon as the robot arrives in each one of it. After all key points have been explored the second run starts and the robot uses the mapped maze with A* to find the goal. Those steps are described in details below.

3.2.1 Storage and mapping

The implementation is not so complex, but it surely wasn't trivial. As commented before, the maze was a discrete grid, so all things related to positions were modeled with a matrix having each grid a row and column. The robot would consider its position and movements based in X and Y position, the matrix index in python, however, don't work in that way. In a n x n grid, for example, robot position [0,0] would mean matrix position [n,0]. In order to consider this it was created a vector that would map the y position in the right index. The implementation can be seen below:

```
# DO A MAPPING TO THE MOVEMENT POSITION TO ACTUAL GRID POSITION
Dimension_vector_mappingX = np.arange(0, maze_dim, 1)
self.Dimension_vector_mappingY = np.transpose(Dimension_vector_mappingX)[::-1]
```

The mapping was discussed in the algorithms and techniques part, where the sensor data was used to model different obstacles matrix for the calculation of the fastest path. Its implementation can be seen below. One clearly sees the Y mapping in the index and how the sensor data are summed or subtracted (depending of the orientation) to save the data in the barrier matrixes.

```
# X coordinate is Y barrier place and Y coordinate is X barrier place with mapping
if self.heading == 'up':

    self.barriers[0][self.Dimension_vector_mappingY[self.location[1]]-(sensors[1]),self.location[0]] = 1
    self.barriers[1][self.Dimension_vector_mappingY[self.location[1]],self.location[0]-( sensors[0])] = 1
    self.barriers[3][ self.Dimension_vector_mappingY[self.location[1]] ,self.location[0]+ sensors[2]] = 1

if self.heading == 'left':

    self.barriers[0][self.Dimension_vector_mappingY[self.location[1]]- sensors[2],self.location[0] ] = 1
    self.barriers[1][self.Dimension_vector_mappingY[self.location[1]],self.location[0]-(sensors[1])] = 1
    self.barriers[2][self.Dimension_vector_mappingY[self.location[1]-( sensors[0])],self.location[0]] = 1
if self.heading == 'down':

    self.barriers[3][self.Dimension_vector_mappingY[self.location[1]],self.location[0]+( sensors[0])] = 1
    self.barriers[1][self.Dimension_vector_mappingY[self.location[1]],self.location[0]- sensors[2] ] = 1
    self.barriers[2][self.Dimension_vector_mappingY[self.location[1]-(sensors[1])],self.location[0]] = 1

if self.heading == 'right':

    self.barriers[0][self.Dimension_vector_mappingY[self.location[1]]-( sensors[0]),self.location[0]] = 1
    self.barriers[3][self.Dimension_vector_mappingY[self.location[1]],self.location[0]+(sensors[1])] = 1
    self.barriers[2][self.Dimension_vector_mappingY[self.location[1] - sensors[2]],self.location[0]] = 1
```

A variable 'goal' was used to specify the goal to which the path algorithm should take. During the exploration run that goal changes between the center of maze and the key points used.

3.2.2 A* and Heuristic

Also in the initialization it was created the heuristic function for the A* method. It is created in two loops, taking in consideration the sum of each contribution for X and Y for the heuristic for an arbitrary maze size and goal. That heuristic is changed every time that the robot change the key point to explore. Its function can be seen below:

```
def Get_heuristic(self):
    for idxX in range(1,self.maze_dim):
        self.heuristic[idxX, 0] = np.absolute(self.goal[0] - idxX) #+ maze_dim/2
    for idxY in range(1,self.maze_dim):
        self.heuristic[:, idxY] = self.heuristic[:, 0] + np.absolute((self.goal[1] - idxY))
```

Then the next direction to the actual goal is obtained with the A* algorithm. The expansion part can be seen below, one can clearly see how the obstacle barriers are used, preventing the expansion in the direction where there is a barrier.

```
Actual_position = To_explore.pop()
x = Actual_position[2]
y = Actual_position[3]
g = Actual_position[1]
h = Actual_position[0]


if x == goal[0] and y == goal[1]: # If goal
    is_goal = True
    Grid_values[self.Dimension_vector_mappingY[y], x] = te
else:
    for idx in range(len(self.movements)): # Loop to see each movement
        x2 = x + self.movements[idx][0]
        y2 = y + self.movements[idx][1]

        # All the conditions that would make the move invalid
        if x2 >= 0 and x2 < self.maze_dim and y2 >= 0 and y2 < self.maze_dim:
            if Visited[self.Dimension_vector_mappingY[y2]][x2] == 0 and \
                        self.barriers[idx][self.Dimension_vector_mappingY[y],x] == 0 : # Each direc
                g2 = g + 1
                h2 = g2 + self.heuristic[x2,y2]
                To_explore.append([ h2,g2, x2, y2])
                Visited[self.Dimension_vector_mappingY[y2]][x2] = 1
                Grid_values[self.Dimension_vector_mappingY[y2],x2] = te
                te += 1

                #print 'te: ' + str(te) + '. x2: ' + str(x2) + '. y2: ' + str(y2)
                #print 'Saved position: ' +str(self.Dimension_vector_mappingY[y2]) + ' '  + str(x2)
                #print 'Grid values' + str(Grid_values[self.Dimension_vector_mappingY[y2],x2])
ialize variables to get next path
 'Grid Values'
```

After all the expansion is done it goes back recursively to the start to find the actual movement needed, the number of repeated movements are saved in memory to see if the intensity of movement should be higher than 1. That can be seen below:

```
while te != 0: # Loop until get from goal to initial position
    x = x2
    y = y2
    Directions = [999, 999, 999, 999] # Start with big numbers
    Positions = [[0, 0], [0, 0], [0, 0], [0, 0]]
    for idx in range(len(self.movements)):

        x2 = x + self.movements[idx][0]
        y2 = y + self.movements[idx][1]

        #print 'Barrier: ' + str(self.barriers[0])
        if x2 >= 0 and x2 < self.maze_dim and y2 >= 0 and y2 < self.maze_dim and \
                        self.barriers[Movment_to_save[idx]][self.Dimension_vector_mappingY[y2],x2] == 0 \
                        and Grid_values[self.Dimension_vector_mappingY[y2], x2] > -1:
            Directions[idx] = Grid_values[self.Dimension_vector_mappingY[y2]][x2]
            Positions[idx] = [self.Dimension_vector_mappingY[y2], x2]

    Movement = np.argmin(np.array(Directions)) # Get the movement of the small number

    # Increase intensity for repeated movements
    if Movment_to_save[Movement] == Movement_memory:
        Intensity += 1
    else:
        Intensity = 1
    # Save in memory
    Movement_memory = Movment_to_save[Movement]


    # Change initial position to position with best movement
    x2 = x + self.movements[Movement][0]
    y2 = y + self.movements[Movement][1]
    te = Grid_values[Positions[Movement][0],Positions[Movement][1]]
    #print 'Grid Values'
    #print str(Grid_values)
```

3.2.3 Mapping Directions

That direction then goes to a function that converts the wished direction to a robot movement, based in his actual orientation, Therefore the importance of mapping orientation. Some lines of the mapping movement function and orientation track:

```
# Map the actual robot orientation and movement to make the right action
def Convert_stage(self,Direction,Intensity):

    ######### Heading Up ##########
    if Direction == 0 and self.heading == 'up':
        movement = Intensity
        rotation = 0

    if Direction == 1 and self.heading == 'up':
        movement = Intensity
        rotation = -90

    if Direction == 2 and self.heading == 'up':
        movement = -Intensity
        rotation = 0

    if Direction == 3 and self.heading == 'up':
        movement = Intensity
        rotation = 90

    ############# Heading left #############
    if Direction == 0 and self.heading == 'left':
        movement = Intensity
        rotation = 90
```

```
######################### HEADING AND LOCATION AFTER MOVEMENT###################
# handle the heading
if self.heading == 'up' and rotation == 90:
    self.heading = 'right'

elif self.heading == 'up' and rotation == -90:
    self.heading = 'left'

elif self.heading == 'right' and rotation == 90:
    self.heading = 'down'
elif self.heading == 'right' and rotation == -90:
    self.heading = 'up'

elif self.heading == 'down' and rotation == 90:
    self.heading = 'left'
elif self.heading == 'down' and rotation == -90:
    self.heading = 'right'

elif self.heading == 'left' and rotation == 90:
    self.heading = 'up'
elif self.heading == 'left' and rotation == -90:
    self.heading = 'down'
# Handle the location
print 'New Heading:' + str(self.heading)
```

The new position and orientation are saved and the output is given. After it finds the first key point a new goal appears for exploration purposes until it comes back to the origin and start the second run using the A* algorithm in the mapped maze.

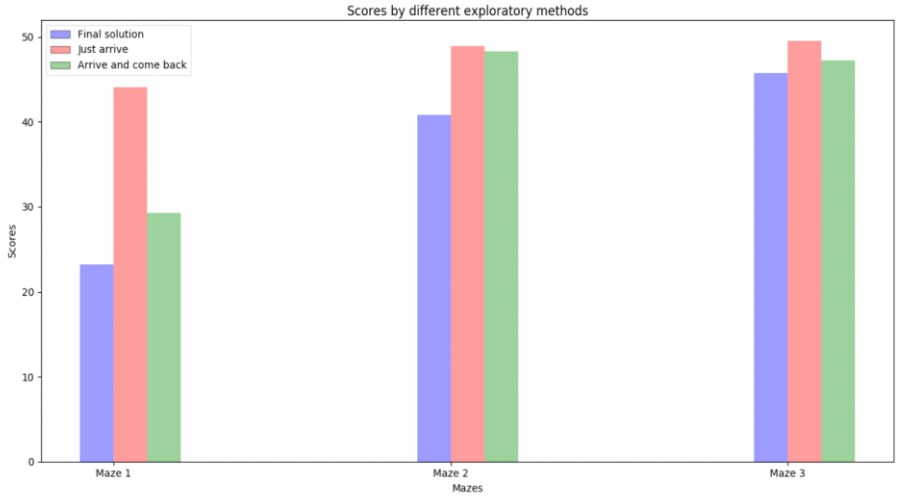
3.3 Refinement

The first thing that was tried was to just arrive at the center with A* and then start the second run. The number of training steps was very low, but the general result wasn't so high in comparison to the other methods. Than it was tried a more explanatory approach in which the agent would return to the start after the goal was founded. The results improved, but could still be better. The last one was the most exploratory of all three, which was discussed before, where the agent would do a swing search with center and corners to find most of the ways that would lead to it, the used key points were the three corners of the maze(up-left, up-right, down-right). In the figure below one can see a comparison of scores between the 3 explorations. The index can be read:

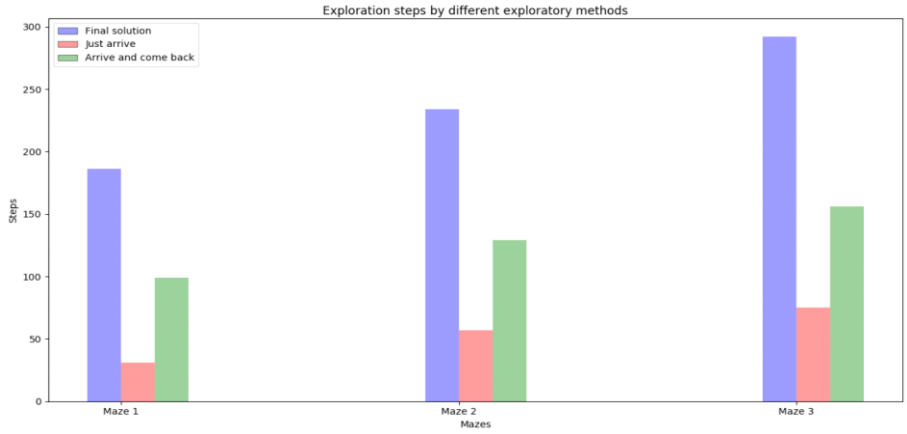
- Final solution : All key points exploration

- Just arrive: A* to arrive and then start the second run

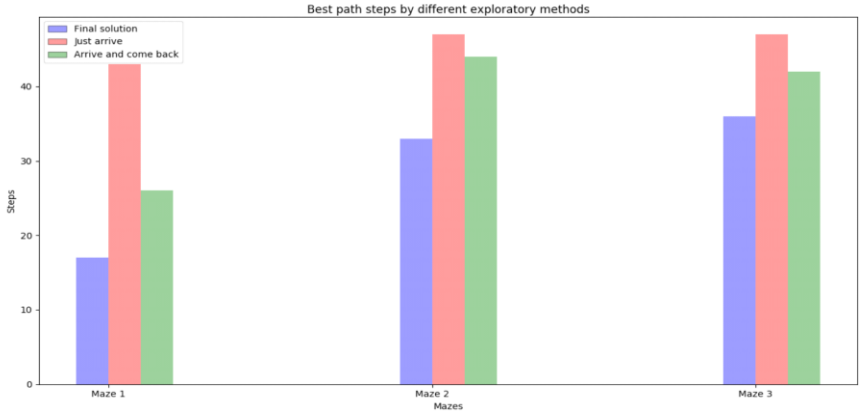
- Arrive and come back: A* to arrive, come back to the start and then start a new run.



The increase in the exploration steps can be seen below:



The number of steps in the second run, however, decreases a lot with more exploration, which is what is expected, as seen in the figure below.



The lines where the key points are implemented can be seen below. After the goal turns back to the start the code start the next run:

```
if self.location == self.goal and self.goal != (0,0):
    # Re start the goal to go back with another way
    self.run += 1
    if self.run == 6:
        self.goal = [0,0]
    if  self.run == 3 or self.run == 5 or self.run == 7: #self.run == 1
        self.goal = [self.maze_dim/2 - 1, self.maze_dim/2]#[self.maze_dim-1,self.maze_dim-1]
    if self.run == 1:
        self.goal = [0,self.maze_dim-1]
    if self.run == 2:
        self.goal = [self.maze_dim-1,0]
    if self.run == 4:
        self.goal = [self.maze_dim-1,self.maze_dim-1]

if self.location == self.goal:

    # Re start parameters
    self.location = [0,0]
    self.goal = [self.maze_dim/2 - 1, self.maze_dim/2]
    self.heading = 'up'
    self.is_exploring = False
    output = ('Reset', 'Reset')
    self.Get_heuristic()
    self.PathMap = np.chararray((self.maze_dim, self.maze_dim))
    self.PathMap[:] = '-'
    self.PathMap[self.Dimension_vector_mappingY[self.maze_dim/2] , self.maze_dim/2-1] = 'X'
    return output
```

- # Results

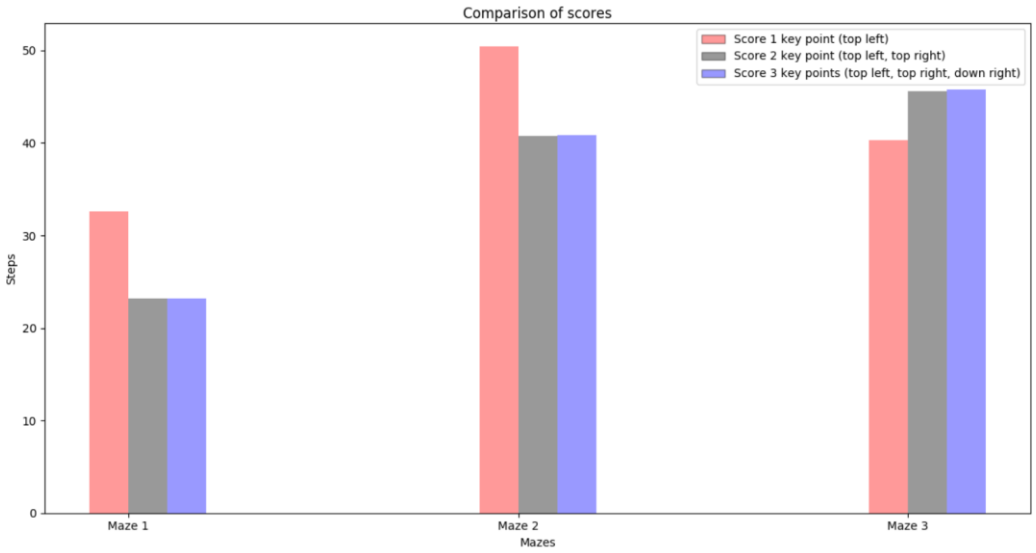
4.1 Model Evaluation and Validation

The developed algorithm can find a good path in all three provided mazes, and, as seen in the refinement section, is better than the other solutions.

The developed robot does exactly what expected in the beginning introductory part, it can robustly map its environment, not hitting the wall any single time and being able to map any wall that it sees. It can well explore the maze using a swing search in between the corners (key points) and the center and with the use of square distance as heuristic in A* it can find the best path of the mapped maze.

We saw that using extra key points to map the maze gives far better results than just going straight to the center and start a new run. An analysis to the number of key points was done, and was found that, in general, 2 to 3 key points is enough to a good result. The number used was 3, because it just performed a little slightly worse than using 2, and allows a better generalization depending of the maze. The analysis

result can be seen below. The only exception was the maze 3, because due to the robot motion, the shortest path doesn't necessary means a shorter number of steps (going 3 squares up takes 1 step while going right, up, right takes 3):



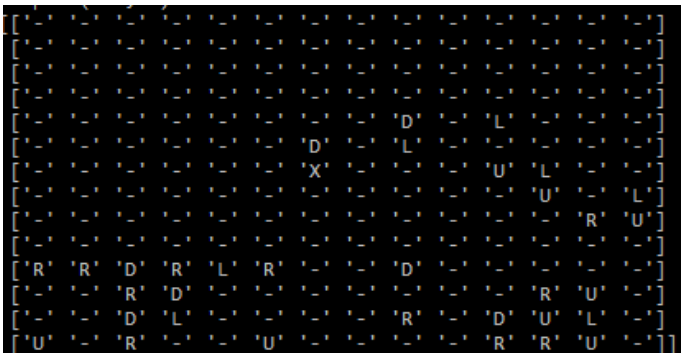
The algorithm was robust to all mazes and also to the created one that will be discussed in the next section, it works well even with new perturbations (obstacles) and, since it does find the goal every time, the result model can be trusted.

The founded path for each maze is given below. The movements are represented as the first letter of its respectively direction 'U' for Up and so on. The X marks the goal and only the places where the robot made a move has a representation, which means one can also see how much steps in that direction the robot took.

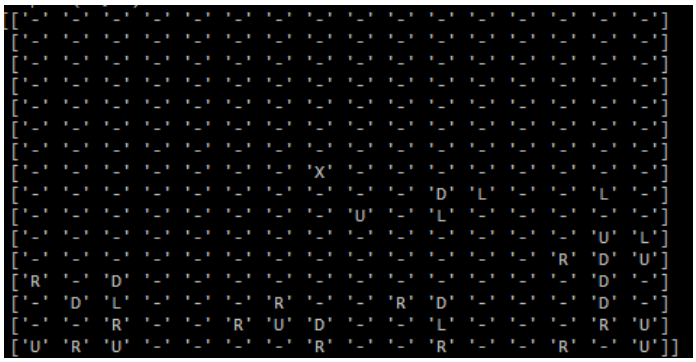
Maze 1

```
[['-' '-' '-' '-' '-' '-' '-' '-' '-' '-' '-' '-' '-']
 ['-' '-' '-' '-' '-' '-' '-' '-' '-' '-' '-' '-' '-']
 ['-' '-' '-' '-' '-' '-' '-' '-' '-' '-' '-' '-' '-']
 ['-' '-' '-' '-' '-' '-' '-' '-' '-' '-' '-' '-' '-']
 ['-' '-' '-' '-' '-' '-' '-' '-' '-' '-' '-' '-' '-']
 ['-' '-' '-' '-' '-' 'X' '-' 'L' '-' '-' '-' '-' '-']
 ['-' '-' '-' '-' '-' '-' '-' 'U' 'L' '-' '-' '-' '-']
 ['-' '-' '-' '-' '-' '-' '-' '-' '-' '-' '-' '-' '-']
 ['-' '-' '-' '-' '-' '-' '-' '-' 'U' '-' '-' '-' 'L']
 ['R' 'D' '-' '-' 'R' '-' 'D' '-' '-' '-' '-' '-' '-']
 ['-' '-' '-' '-' '-' '-' 'R' 'D' '-' '-' '-' '-' '-']
 ['U' 'R' '-' '-' 'U' '-' '-' 'R' '-' '-' 'R' 'U' 'U']]
```

Maze 2



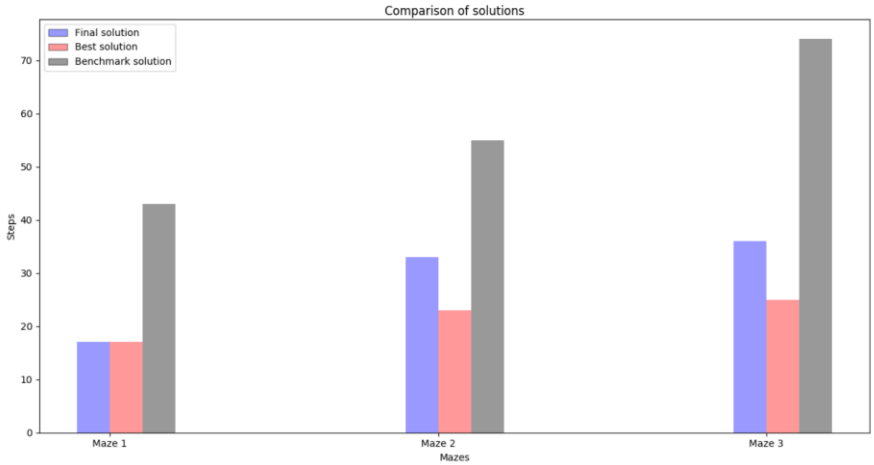
Maze 3



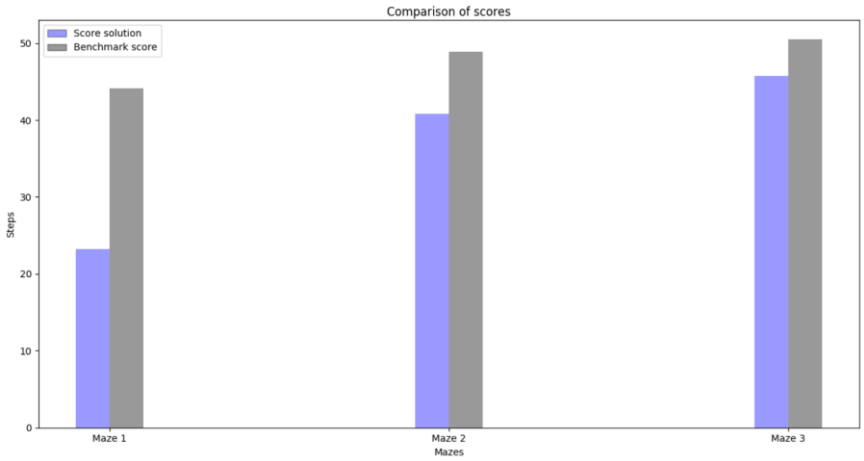
4.2 Justification

The algorithm score is compared with the proposed benchmark and then with the real path, so one is able to see how far it is for the perfection and how better it is than the benchmark:

Comparison of solutions



Comparison of scores



The algorithm is far better than the benchmark in the real path and score and, for the maze number 1, the solution founded was the best. For the maze 2 and 3 there's still some difference due to not enough exploration, a solution would be a better algorithm of exploration or a higher number of key points to explore, that second one, however, could substantially increase the exploration steps.

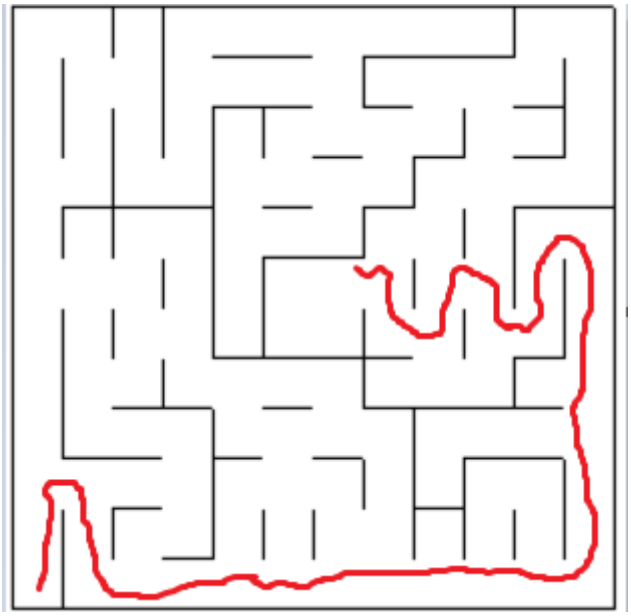
Since the algorithm is robustly better than the benchmark, can every time reach the goal in every maze and, in some cases, can even find the best solution, one can safely say that the problem was rightly solved.

Results

5.1 Free-Form Visualization

To show the efficiency of the algorithm, the first maze was modified in a way that there´s two main ways to the center, the first longer one being from up and then right/down and the second shortest one being right, up and left. In this maze, due to the heuristic function, the A* algorithm alone would go to the first longer one because of the heuristic function, since going to the faster way would mean to deviate too much from the goal. The developed algorithm, though, can correctly find the best main path in this maze, with a score of 25.93, being 207 exploration steps and 19 real steps.

The developed maze can be seen in the figure below with the fastest path and the path founded by the algorithm for both the solution and with just A*:



```
[['-' '-' '-' '-' '-' '-' '-' '-' '-' '-' '-' '-']
 ['-' '-' '-' '-' '-' '-' '-' '-' '-' '-' '-' '-']
 ['-' '-' '-' '-' '-' '-' '-' '-' '-' '-' '-' '-']
 ['-' '-' '-' '-' '-' '-' '-' '-' '-' '-' '-' '-']
 ['-' '-' '-' '-' '-' '-' '-' '-' '-' '-' 'D' 'L']
 ['-' '-' '-' '-' '-' 'X' '-' 'L' 'D' 'L' '-' 'U']
 ['-' '-' '-' '-' '-' '-' '-' 'U' 'L' 'U' 'L' '-']
 ['-' '-' '-' '-' '-' '-' '-' '-' '-' '-' '-' '-']
 ['-' '-' '-' '-' '-' '-' '-' '-' '-' '-' '-' 'U']
 ['R' 'D' '-' '-' '-' '-' '-' '-' '-' '-' '-' '-']
 ['-' '-' '-' '-' '-' '-' '-' '-' '-' '-' '-' '-']
 ['U' 'R' '-' '-' 'R' '-' '-' 'R' '-' '-' 'R' 'U']]
```

5.2 Reflection

The problem was adequately solved, the robot is able to map the environment using its sensor data as a way of store wall positions, visit key points of the maze for exploration and successfully move itself with the A* algorithm. in the beginning it was very difficult to come out with a way to map the walls in a way that I could later use with A*. I thought at first about a matrix that would turn depending of the direction, but ended up finding the solution for the four matrixes, the index, however, was bugging me for a while, doing the vector mapping was a so simple solution that afterwards I couldn't believed it took so long.

I had the same feeling with the movement intensity, saving repeated moves in a counter is so simple but took me a while to actually

think in it. I believe that the A* implementation is very clear, it runs very fast and the code is very concise. The heuristic, however, could be better, because I only take the squares distance as heuristic, not the robot movement intensity, so the function has no preferences for straight paths (which would be faster).

5.3 Improvement

A possible improvement for this code, as commented before, would be a heuristic function that maps not only position but also steps, so it would prefer to go to ways were it can do more simultaneously steps reducing the time.

The problem solved here was in a discrete world, which is not the case in many real applications. Depending of the problem we can, however, discretize the problem, if or robot would have a radius of 0.4 units, the walls being 0.1 units thick, it is still possible to discretize the domain working for the robot to not hit the wall based in some sensor metric. Like implementing in code a continuous sensor value to model both a discrete grid position and a maximum distance from the wall metric, allowing to still be able to use grids but taking care about the added complexity with a PID control. The most difficult case I can think of would be some very curve labyrinth, which would ask for a more discretized grid to work properly.

Having a continuous domain would allow one to also use a smooth path transition. In the grid world we consider just very 'hard' movements. This is not always the case in a car, for example. Thus being able to smooth movements would be also a great improvement.

Another point that wasn't considered here was the uncertainty of sensors. The work was done in a deterministic world, if it was to handle uncertainties, some algorithm like Kalman filters and particle filters could be used to have a nice position distribution which would be able to work even in noisy sensor environments.