RMI report

Pedro Gonçalves[88859] and Pedro Silva[89228]

Departamento de Eletrónica, Telecomunicações e Informática

Universidade de Aveiro, 3800 Aveiro, Portugal

Abstract. In this report we will describe the different approaches and logic implemented to complete each of the 3 different tasks (localization, mapping and planning) proposed on the second assignment of RMI. This implementation is based on the given objectives and the different types of data (obstacle sensors' values and compass). To move the robot, we send a speed value for each of the 2 robot motors. The first task we had to complete was to make it possible for the robot to locate itself in the maze, since we didn't have GPS in this assignment and the compass, motors and sensors are all noisy, this was particularly hard. The second task was to complete the maze in order to completely map the navigable cells and find the beacon's position, and then return to the starting spot. Finally, for the third and last task, we had to compute a closed path with minimal cost, that allows us to visit all the beacons, starting and ending in the starting spot. Both the maze and the final path have to be written in two different text files.

1 Problem identification

In this assignment we developed a robotic agent to command a simulated mobile robot in order to locate himself in the maze, map the entirety of the maze and plan the best closed path between the origin and all the beacons. We had access to a noisy compass, with a standard deviation of 2º, to the 4 obstacle sensors, situated in the front, left, right and back of the robot and also, to the 2 motors of the robot. The robot was then placed in an arena with a maximum size of 7-cells tall and 14-cells wide. In this arena, there are vertical and horizontal walls, with a width equal to 0.2 robot diameters. These walls, as well as the arena limits can be detected by the obstacle sensors.

2 Localization

The first thing we did was changing the right and left sensor's location to 90º and -90º, so that they would all have a 90º angle.

2.1 Movement model

Our goal was to make the robot move from the center of one cell to another, these cells are always on the even positions of the map.

For the movement of the robot, we had to take into account the inertial characteristics of the motors. Therefore, we calculated the moved distance with the following equation:

$$out_t = \frac{in_i + out_{t-1}}{2}$$

Fig 1. Equation to get the power applied to the motors.

In this equation, in_i is the power applied to the motor, out_t - 1 is the motor speed in the previous cycle and out_t will be the motor speed in the present cycle.

We have a sum variable (self.sum) that is refreshed at every cycle with the addition of the out_t value. To be fast and precise using this equation, the motors receive full power (0.15) and after a certain self.sum value, reduce the speed to be able to stop when the sum is 2. We need to make sure that the sum is 2 because that's the distance between the center of 2 cells.

The robot motors also have noise, so the robot orientation wouldn't always be the desired one. Our way to solve that problem was to read the compass values at each cycle. If the values were between the desired angle (0º, 90º, -90º, 180º) plus a deviation of 2º for the compass noise we wouldn't change the robot orientation, if the orientation had a bigger deviation than 2º and smaller or equal than 5º, we would make a slight adjustment to the robot motors, and if the deviation was between 6º and 15º we would make a bigger adjustment to the motors to make the robot go back to the desired orientation. For example if we wanted the robot to go in 90º and the compass gives 96º, we send the left motor a slightly lower value in order for the robot to adjust his orientation.

The obstacle sensors are also used to see if the robot is too close to the left, right or front wall. If it is close to the left or right wall, we use the same correction that we used on the compass, in this case not only to adjust his orientation, but also to move a little bit away from the close wall. The front sensor is used when the noise is a bit bigger, and the movement equation doesn't work exactly as expected, which leads to the robot getting too close to the front wall. In this case, the robot reads the front sensor's value and stops before reaching the 2 units of the sum and avoiding the collision with the wall.

2.2 Turn model

One of the most important things that the robot has to do is turn to a specific orientation. In order to achieve that, we use the compass. When the robot has to turn 90º, it first stops in the middle of the cell, and then the motors are given an opposite value of power, so if we want to rotate right, the right motor is given a negative value and the left motor, the same value but positive, and if we want to turn left, we do the other way around. When the robot has almost fully rotated to where it wants, we reduce the power applied to the motors in order for the rotation to become slower. Finally, when we read 3 straight values with the desired orientation with the standard deviation of 2º, we know that the robot has turned to the right orientation and is ready to go in front.

3 Mapping

On the second task, we had to map the entire maze to a text file, as well as the beacons split across the map and finally send the robot back to the starting spot.

3.1 Map discovery

To start off, we gave the robot the following priorities: left, front, right, back. This means that if there's an available square on the left, the robot will always go there (unless it's already been there), if it's not, it will go in front, then it will go to the right and finally, if no other option is available, it will go back.

This strategy is good in the beginning of the map, but it's easy to realize that sooner or later the robot will drive in circles. To avoid this, we implemented Astar.

3.2 A*

First thing we did was to create a list with the squares the robot hasn't been on yet. Every time the robot gets to an intersection, it saves the free squares on the *squares_to_visit* list.

The next time the robot is about to go to an already visited position, it calculates the path to the closest unvisited position instead, and goes there.

Finally, when all positions are visited, we use A star once again to send the robot to the starting position.

3.3 Print the map

In order to print the map correctly, we read the 4 sensors of the robot on every position and mark the walls with a '-' or '|' if it's a horizontal or vertical wall, respectively, and with an 'X' if it's a free square. This way, every time the robot moves to another square, the map is updated.

Meanwhile, we save each beacon in a dictionary, and when the robot completes the map, we print the entire map to a text file and override the 'X' positions where there's a beacon with the correct beacon number. For example:

```
 - - - - - - - - - - - - - -
|XXXXXXX|XXXXXXXXX|XXXXXXXXX|
 - - X - X - - - X X - - - X
|XX0XXXXXX|XXXXX|X|XXXXXXXXX|
 X - - - X X - - X - X - - X
|XXX|XXX|XXX|XXX|XXX|XXX|3XX|
 - X X - X - X X - X - X - X
  |X|XXXXX|XXX|X|1|XXXXX|XXX|
 - X - - - X - X X - X X - X
|XXX|XXXXXXX|X|X|XXXXX|XXXXX|
 X - X - - - X X X - - X - X
|X|2|X|XXXXXXX|XXX|XXXXXXXXX|
 X X X X - - X - - X - - - X
|XXXXXXXXXXX|XXXXXXXXXXXXXXX|
 - - - - - - - - - - - - - -
```

4 Planning

For our third and last task, we had to compute a closed path with minimal cost that allows the robot to visit all target spots, starting and ending in the starting spot.

4.1 Implementation

In order to do that, we needed to do 3 things, first we had to discover all the beacons, then we had to calculate all permutations that allowed us to make the path, starting and ending in the starting spot, and finally we had to print the path with the minimal cost to a text file.

4.2 Permutations

To make the necessary permutations, we used a library called itertools, which has a method that calculates the permutations, given a list. In this case, we have 3 beacons, since the beacon 0 is the start and end point so it doesn't affect the permutations.

To calculate the permutations, we used a python library called itertools, in which we passed a list, in this case [1, 2, 3] (list with 3 beacons), and found all the permutations with these numbers.

```
nBeacons = 3
permutations_ls = []
for j in range(1, nBeacons + 1):
    permutations_ls.append(j)

ls = list(itertools.permutations(permutations_ls))
print(ls)
```

```
[(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]
```

After finding all the permutations for the correct number of beacons, we ran through that list and calculated the path which starts and ends in the starting point and has the minimal cost, using A-star.

4.3 Print the path

Finally, after finding the closed path with minimal cost, we printed the path to a text file.

5 Results

The results were satisfactory, the robot maps the entire map and plans the best path, 9 out of 10 times. However, when it fails, it is usually because it gets too close or too far from an arena border and ends up colliding or misreading the sensors because of their unusual values, since on the border of the map, the walls have no diameter. That happens because sometimes the robot is not really centered because of the noise, and after that moment it might fail. It's unusual, but can happen. We had some problems with the A* computation time, so it was implemented a wait system, in order for the A* to be completed and the robot to follow the right path.

6 Conclusion

To conclude, it was hard to find a way to make the robot stay in the center of the cells, because of the motor's noise. Our way of advancing the 2 units might not have been the best, since we stopped every 2 units, even when the path is straight for several units. With that approach, we can't always complete all the mapping in under 5000 tick cycles.

References

1. https://www.laurentluce.com/posts/solving-mazes-using-python-simple-recursivity-and-a-search/, last accessed 2021/11/15.