Robotic challenge solver using the CiberRato simulation environment

Bruno Aguiar, n.mec.: 80177, and José Moreira, n.mec.:79671

University of Aveiro, Aveiro, Portugal

Abstract. Report about the Robotic challenge solver using the Ciber-Rato simulation environment.

Keywords: Robotics · Ciber-Rato · Control · Mapping · Planning

1 Introduction

The following work is a report on a project regarding the class *Robótica Móvel e Inteligente*, where we were asked to use the CiberRato and complete some challenges with the provided tools. The goal is to comprehend how the robot works, what it takes from it to move and how to make it explore a maze while reporting everything it sees.

2 C1 – Control challenge

2.1 Deciding logic

The C1 challenge was completed with a simple approach. If the robot had room to move forward, it would. As it was moving, measurements of the side sensors were constantly taken. If the robot was closer to a wall on its left side, an adjustment was to be made, that would change with how close to the wall the robot was. If it was really close, an aggressive direction change would take place to move the robot away from the wall, if it was approaching a wall but not really close, a lighter rotation would be made. The same logic applies to when the robot is close to a wall on its right side. If the robot faced a wall on its front, the side sensors were used again, rotating to the left side if there is no wall there, or to the right side. The goal of 10 laps to this challenge was not completed in its fullest, as the robot can only complete around 6 or 7 laps.

3 C2 – Mapping challenge

In the challenge C2, our robotic agent can be modeled into five states: `choose` , `walk`, `rot_left`, `rot_right`, and `end`. Before jumping into the first state, its initialized a quadruple matrix in order to store all the walls and passages. It's unknown the location and orientation of the map *a priori*, so the agent will

assume that it's in the center of the quadruple. The size of the quadruple is given by `MAPPINGCOLS` and `MAPPINGROWS`, and the agent will start in the position `(MAPPINGROWS/2, MAPPINGCOLS/2)`, will check its neighborhood for passages and walls, creating lists of walkable neighbors, and will update its state to `choose`.

On `choose`, it will compute the fastest path to the closest cells yet to explore using `A* Search Algorithm`. Both the tree search file and the use on the main file came from another class we had, namely Introdução à Inteligencia Artificial years ago, from our project Bomberman. Among other search algorithms like the depth search and greedy search. For the depth search, this algorithm falls often into infinite loops, because it picks the deepest node first. We were unsure to use the greedy search or the A* Search algorithm, due to both using heuristics to calculate the best path. Although we went to A* because greedy algorithm does not store the cost from the initial position (node) to the current node and A* gave us better results overall. Based on the generated path tree, the agent will starting picking the path tree nodes by order, following that path. The initial node is going to be the current cell position of the agent in the map and the goal node is going to be the first cell of the sorted list of cells known but yet unexplored. The sorted mechanism uses the euclidean distance to compute the distance between two cells and if there's an unexplored cell right on its front, or on its sides, it gives the highest priority.

The chosen cell is compared to the current cell, and, based on the orientation of the two cells, the `choose` state will tell the agent who its next state is going to be: `rot_left`, `rot_right`, or `walk`.

In the `walk` state, we instruct the robot to move a cell. First, we find the direction the robot was headed, since the robot always moves forward. After that, the robot moves accordingly. Assuming the robot is facing North, to move forward, the GPS coordinate that will change is x. We compare the position the robot is to the position the robot is instructed to go, if that position is reached, the robot goes bock to the `choose` state, if not, it moves forward. It also goes back to that state if it about to hit a wall facing forward. To try to keep the robot centered in the cell, we used a Proportional Control, *PControl*, However, sometimes that was not enough. To resolve that, we used the side sensors, and whenever the robot was closer to a wall, a little tweak in the direction was made. Most of the times, when the robot is fairly centered, the *PControl* was enough.

In the `rot_right` state, much like in the walk, we identified the direction that the robot was facing to facilitate the rotation. Again using the example where the robot was facing North, if the robot was said to rotate 90 degrees, we would identified an interval where it would be acceptable for the robot to stop rotating. In this case, as the perfect angle to stop rotating was 90 degrees, we used the interval between -89 and -91 as acceptable. Since the robot was rotating right, it would approach the goal from -180 to -90. to accelerate the process, while the compass measured more than -65 it wold rotate faster, and from that to the

goal it would rotate much slower to not over rotate. This state also allows 180 degrees rotation, to when the robot has to change direction.

The `rot_left` state is in everything equal to the previous state apart from the ability to rotate 180 degrees, since it is not necessary whit that being already available.

The goal to this challenge was achieved. However in some cases, because of the noise in the sensors, some errors may occur, even though most of the times the map is fully discovered and correctly printed to the output file. The error occurs when the robot missteps a cell, identifying the center of the cell while it is in fact on the start of it. When the robot is again instructed to move forward and daces a wall, it won't pass or hit that was, but it will mar another cell, mapping two cells where it was supposed to be one. This changes the whole map, making the robot spin, and making it impossible to finish the challenge, however, as stated before, most of the times the robot con complete it with no problem and with some time left.

4 C3 - Shortest Path with Beacon Discovery

In this challenge, most of the previous challenge was used to discover the map. he states are pretty much the same, and the only difference is that in the `choose` state we used the measure ground to identify if we are stepping on a Beacon, since every normal ground apart from Beacons return -1. As we explore, with the same methodology as C2, we store the coordinates of beacons as we pass by them. This state is always comparing the number of discovered beacons to the total number of beacons in the map. When this number match, meaning we found every beacon, the robot spots and calculates the path from beacon 1 to beacon 2 and from beacon 2 to beacon 3. The paths are then concatenated and written in to a file.

5 Conclusion

The main goal of this assignment, at least for was, was completed, since it gave us a brief idea of how to control a robot. Regarding the first challenge, where the objective was to lap the maze 10 times, it was only partly completed, as the robot was only capable to do around 6 to 7 laps, mostly because it was the beginning of the project and we were not sure how everything worked, however we could not visit it again due to time constrains. The second challenge was completed. The robot can discover the whole map, while printing it exactly the way it is intended with some time left. Has some issues with noise already mentioned but overall was a success. In the third challenge, the robot can find all the beacons and calculate the path from beacon 1 to beacon 2. However, mostly because of the depth of the tree, or at least is what we suspect is the issue, it can not calculate the full path from beacon 2 to beacon 3, meaning this challenge is done, but uncompleted.