Pathfinder solver using MicroRato

Pedro Santos (76532) and Ruben Kins (92078)

Universidade de Aveiro, Aveiro, Portugal

Abstract. This document summarizes the development of a robotic agent. The goal of the robot is to explore a maze and find a token. After finding the token the robot has to return to the initial position using the fastest path available. For the return the robot uses a map as a result of the previous exploration. The project was a success and the robot could master all the challenges given in the example course.

Keywords: Robotic Agent · SLAM · C · A* · pathfinding

1 Project Development

The world the robot has to explore consists of a white wooden plate with black tape on it. The black tape defines a maze that the robot has to navigate. The maze consists of equidistant cells that form either intersections or straight segments. The project can be divided into several sub-tasks that need to be implemented. The challenge is for the robot to simultaneously explore and navigate in the world. The project is divided into several sub-tasks:

1. Locomotion: The robot should be able to follow the black lines and execute turns, for example to turn right at an intersection.
2. Sensing and Localization: The robot perceives the environment only through ground sensors that distinguish between black and white. Additionally, it relies on internal odometry.
3. Map building: The robot should explore its surroundings and store the results in a map.
4. Exploration: The robot needs to collect information about the world in order to navigate efficiently and find an efficient path towards the goal
5. Pathfinding: After finding the token it is critical to find an efficient path back to the starting position.

For this project a hybrid architecture of reactive and deliberative behavior is used. The locomotion part outputs motor commands that are a direct result of sensor input and thus shows reactive behavior. As soon as an intersection or the end of a straight segment is reached the locomotion part expects commands from the higher, deliberative functions about where to move.

2 Locomotion

The locomotion is one of the most basic parts. The input is the measurement of the three central ground sensors and the output is the motor command. Generally, the robot moves forward. The forward movement is in superposition with turning movements that aim at centering the robot over the black line. Depending on how far offset the robot is the turning movement is slow or fast (cf. Fig. 1). In order to remove effects from inertia and generally make the robot move

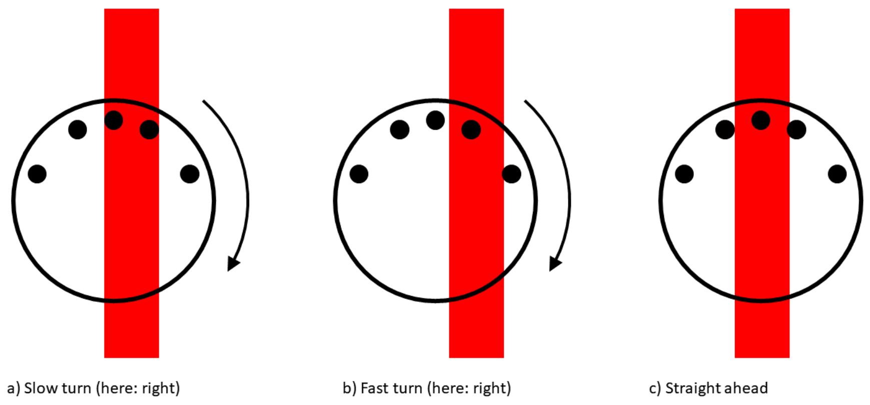

Fig. 1. Turning behavior of the robot.

smoother the motor command is not applied directly but applied to a buffer. The effective command is the average of all buffer entries. In this case a buffer size of three at a cycle time of 40 ms is a good trade-off between smooth movement and delayed reaction time.

3 Localization and Sensing

The challenge of localizing the robot is that there is no direct source of information like a compass. The robot can rely on its odometry, but after a while errors accumulate and the estimation becomes unreliable. For that reason the robot position is reset after finding a node in the map so that odometry is used only as a reference to the last explored node.

Because the map can be represented in a topological fashion as a set of nodes with x- and y-coordinates the robot state reflects that: The robot state is given by discrete x position, discrete y position and discrete orientation (Because the lines are all perpendicular or parallel to each other). Directions can be represented both in local reference (front, back, right, left) and global reference (north, east, south, west). Sensing and movement use local reference while map building and movement commands use global reference. Functions are implemented that translate between both domains using the robots orientation state.

The ground sensors play a critical role in the detection of nodes because they are the only available input. For that reason a function to stabilize the sensor readings is implemented. Its purpose is to prevent random pieces of dirt or sensor noise to trigger a false event. The function consists of a state trigger for each sensor and a buffer. The state trigger only changes the sensor state if all buffer entries disagree with the current state. For example, if the state is 1, but the buffer reads [1, 0, 0, 0] the state remains 1 because a single entry agrees with the current state. The method causes a delay which is why it is only used for node detection, not locomotion.

There are four different types of node:

1. Straight segment;
2. Dead end;
3. Intersection (includes simple right or left turns);
4. Token (or cheese);

When the robot travelled more than a cell distance and did not detect anything else, it could be a dead end or a straight segment. If any of the central sensors pick up a black line it is a straight segment, else it is a dead end. Distinguishing between an intersection and the token is more difficult because both can be T-shaped. The difference is the line thickness which needs to be measured. As soon as the outer sensors pick up a black line the regular navigation is interrupted and instead the robot moves forward a bit so that it is centered above the node. While centering the robot measures the distance it travels while measuring black on both outer sensors. If the distance is greater than the regular line thickness, the token is detected. If not, a T-shaped intersection is detected. If the front sensor also picks up something, a cross shaped intersection is detected. If along the way only the left or right sensor triggered, there is only a path leading the corresponding direction and maybe a path leading straight ahead. After the paths are detected a flag is set that triggers a map update. The flag includes node coordinates and the available paths.

4 Deliberative Section

This section contains the reasoning behind the developed solution for the robot to be able to find the token and then return to the initial point going the best possible path. The developed agent has 4 main states and each one corresponds to the current navigation goal of the agent:

1. Exploring the map
2. Exploring the map after finding the token

3. Returning to the token
4. Returning to the initial point

In the figure 2 is the state diagram of the agent.

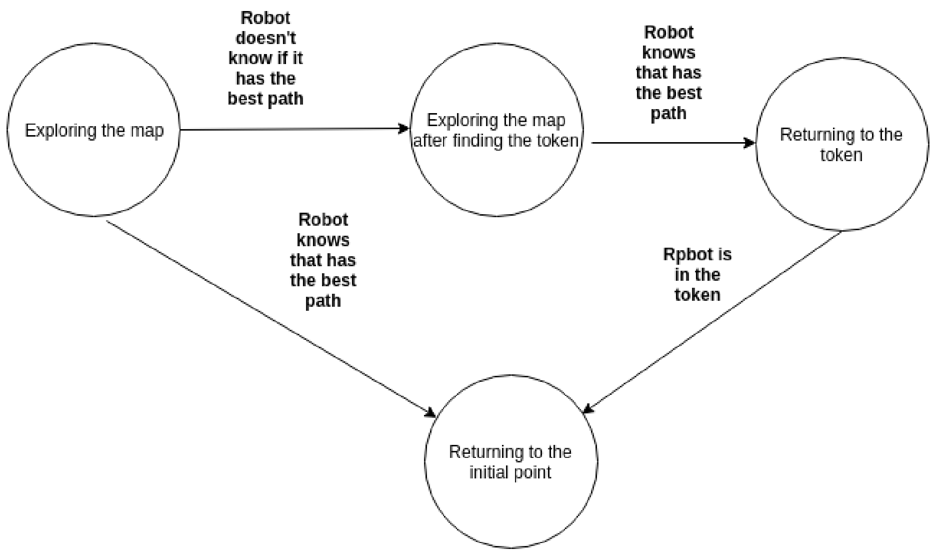
Exploring the map
Robot doesn't know if it has the best path
Exploring the map after finding the token
Robot knows that has the best path
Returning to the token
Robot knows that has the best path
Rpbot is in the token
Returning to the initial point

Fig. 2. State diagram

4.1 Map Building

The first thing that the robot does is initialize the map. The map is a bi-dimensional array containing elements of the type 'Node'. The initial size is 37 in each dimension because the map has at maximum 19 nodes, but the initial position of the robot is not known. Therefore it is necessary to consider the 'worst' case possible which is when the robot is in a corner of the map and the map has the maximum size. The map can extend in every possible direction, so it is necessary to have an array capable of handling four maps due to the fact that it is not possible to know in which sector of the 4x4 map the real maze is. The middle index of this bi-dimensional array is the node (0,0) which is the initial position of the robot. The first element of the array is the node with coordinates (-18,18) and the last one is (18,-18). The members of the structure 'Node' can be seen in the following list:

– int coor_x;
– int coor_y;
– int paths[4];
– int gCost;

- int hCost;
- bool visited;
- struct Node *parent;

The members that are store the dimensions of the map are coor_x and coor_y, which denominate the coordinates of the Node. The array 'paths' is initialized with index -1 in every entry because the paths are unknown. The array is updated accordingly if that node has paths in north, east, west or south direction. For example, if the Node has a path leading to the north the first index of the array is 1 and if it does not have path leading east the second position is 0. The member 'visited' indicates whether the robot already visited the node and the reason behind that is explained in a later paragraph.

The next step after initializing the map is to handle updates to it. When the robot it is in the state of exploration of the map (even after finding the token) it will use the function 'updateMap()'. This function receives two arguments: the current node coordinates and the its paths. The update-function is divided into two parts, one to update the map with the paths and another to update the map size. The first part updates the paths of the current node by populating the array with the corresponding information. However, with the information from the current node it is also possible to update the neighboring nodes. For example, if there is a path to the north it is possible to conclude that the neighboring node in the north has a path leading south. If for example there is no path to the east it is possible to conclude that the eastern neighbor does not have a path leading west. Doing this leads to a much faster and more efficient exploration of the map and search for the best path. The member 'visited' of the current node is also set to 'true'.

The second part of 'updateMap()' has the goal of updating the size of the map. In the beginning the bi-dimensional array is created with all the possible map configurations like explained before. However, if the robot moves upwards for example it is possible to update the size of the map by deleting the last row. To accomplish this it is first verified whether the current node is in a row or column of the array not used before. If that is true it means that the opposite row or column can be deleted. In terms of code if the x-coordinate of the current node is greater than 0 it is possible to delete the first column. If it is less than zero it means that we need to check if is possible to delete the last column of the map array. To do that the initial map size is divided by two and subtracted with the x-coordinate of the last column. If the value of this subtraction is smaller than the x-coordinate of the current node the last column is deleted.

To delete the last column the map size simply needs to be subtracted with 1. To delete the first column it is necessary to shift all the elements of the second dimension of the array. The reasoning for deleting rows is similar. The base point (the node corresponding to (0,0)) also needs to be updated every time a shift is done. This point is necessary in the function 'getNodeMapIndex()' which maps the wanted world node coordinates to the corresponding map array index. Furthermore, when an update is executed the function 'updateUnknownNodesHistory()' is called. This function verifies whether the current node has unknown neighbors and if that is the case it will add it to an array called 'unknownNodesHistory' that saves all nodes with unexplored paths. If the unknown neighbours get discovered later, the node is removed from the list. The use of this array is explained later.

4.2 Exploring the Map

Exploring before finding the token When the robot is in this state in first place verifies if this node has the token. It it has the token will check if the best path is already available using the function checkIfBestPathIsAvailable(). If the best path is available the robot changes its state and it will go to the initial point using the best path calculated with A*, if the best path is not available or it is not possible to ensure that the best path is already calculated the robot will go to the state "Exploring the map after finding the token". The function checkIfBestPathIsAvailable() checks if the best path is already calculated by comparing the size of the path of the A* with the Manhattan distance between the initial point and the token point and if it is the same we can conclude that it is the best path and it is verified if all the nodes of the closed set of A* have known neighbours. If all the nodes of the closed set have known neighbours it means that the best path was found. If the current node doesn't have the token the robot will continue to explore the map using the function getDirectionToExploreMap(). All this functions that returns directions to the robot starts by verifying if it is to follow the A*. For example, if the robot wants to go to a node and performs the A* to that node to known the best possible path and after performing the A* it enters in a state that will follow the A* and only when arrives tho the final node of the path will exit from that state and will have the normal behave. By doing this it is only necessary to calculate once the A* and then it is saved the result, so the performance it is increased, however this is only possible because the world is not dynamic. So, after verifying if it is to follow the A* and if it is not to follow the A* the robot will check if it has a neighbour that has only one path. but it never went there. If it has a neighbour with this properties will go there and this nodes have higher priority to the unknown neighbours. This reasoning is made because it means that it was updated by not going there, but only using their neighbours and there is a higher change of being there the token because it is a dead end and without going to a node it is not possible to known if it has the token. Then, if there isn't neighbours with this properties will go to the unknown neighbour. So, it verifies for each neighbour if it has a unknown neighbour. The behave of the robot must be predictable, so the robot will choose by this order the unknown neighbour North, East, South and West. So, for example every time that the robot has a unknown neighbour at North will choose this one. If the current node doesn't have neighbours that never went there with only one path or unknown neighbours it will be made A* to the nearest node with unknown neighbours. The nearest node with unknown neighbour is the last element of the array updateUnknownNodesHistory and the construction of the array was explained in the end of the section before.

Exploring after finding the token The computational power of the robot is limited, so it is not possible to performer in same cycle a lot of times the A*, so in this state the robot verifies if the last state was ExploringMap and if it was the robot will not check if the best path is available because that verification was made in the end of the last state and this verification runs the A*. Another thing that it is verified is if the robot is following the A*, if the robot is following the A* it is not necessary to call this function, because the robot it is only travelling in known nodes, so it will not change the output of this function. This is done with only one if and the tree conditions are separated with and the last condition is the function, because doing this the function is only called if all the other conditions are true and doing this it is ensured that the A* only runs when it is necessary. So, if the last state was the current state and if it is not performing the A* and if the best path is available the robot will go to the state "Returning to the token", however if this condition is not true will get the direction accordingly with this state. The function is the getDirectionToFindBestPath() and like was explained in the previous state it will verify if it is performing the A*. If it is not following the path of A* the function will verify if it is in a sub state. This sub state consists in calculating the node with unknown neighbours with lower fCost or if there is several minimums the node with a lower hCost of the closed set of the A* between the initial node and the target node. At same time it is verified if the current node continues in the closed set and if it has unknown neighbours. If this condition is true the robot will explore the unknown neighbour of the current node. This is done because it is known that the best path was found when all the nodes of the closed set have known neighbours, so if the current node belongs to the closed set and it has unknown neighbours the best thing to do is continuing exploring this node. If this condition is false it will change the sub state and it will perform the A* to the node with unknown neighbours with lower costs. When it arrives to calculated node, the sub state is different so the robot will check for the unknown neighbour of the current node and will explore that node and the state it is changed again to go to analyse the closed set. This solution is efficient because it can calculate fast the best nodes to search the best path. This ensures that the best path is found because the closed set are the nodes that were analysed by the A* and were discarded because it would not be possible to use them to find the best path, however if this nodes have unknown neighbours maybe they can be used by the best possible path, so it is necessary to explore this nodes. If the closed set doesn't have nodes with unknown neighbours it means that the A* explore all the necessary nodes of the map to find the best possible path. The node with lower cost is chosen because it is a faster approach to go to the more convenient node and the robot does in part the reasoning of the A* in run time.

Returning to the token In this state the agent will go to the token using the path from the A* algorithm, to be able to perform the best possible path from the token to the initial point. When the robot it is in the node with the token will change the state to "Returning to the initial point".

Returning to the initial point This is the last state. The robot it will go from the token to the initial position using the best path possible.

Arrays To handle all the arrays that have nodes from the map was created one structure and a set of functions to handle the manipulation of the arrays. The structure has the following members.

– Node **array;
– size_t used;
– size_t size;

5 Results

The project was an overall success. The robot can independently explore and navigate the maze. The additional challenge in this project was the fact that a real and not a simulated robot was used. Implementing the A* search algorithm proved to be much more challenging using C than it was using a high level language like Python. Improvements can be made by making the code real-time capable. An equal cycle time is achieved with the usage of a simple wait-command, but that disregards the calculation time from the A*-algorithm which causes the robot to trip up sometimes.