Intelligent And Mobile Robotics
Pathfinder solver using the CiberRato simulation environment

Diogo Silva (dbtds@ua.pt)
DETI, University of Aveiro
Aveiro, Portugal

***Resumo* – Este artigo descreve um agente desenvolvido para Robótica Móvel e Inteligente na Universidade de Aveiro. O objetivo principal do robot é de resolver um labirinto, sendo que contém uma área que o agente consegue identificar com a àrea final, após atingir essa área deve voltar para a posição inicial o mais rápido possível. O labirinto contém paredes veriticais e horizontais que o agente deve evitar colidir com ajuda de sensores e atuadores. Este artigo contém a descrição do robot, a lógica usada no código desenvolvido de forma que o agente supera-se o desafio.**

***Abstract* – This paper describes an agent developed for Intelligent and Mobile Robotics at the University of Aveiro. The main objective of the robot is to solve a maze, which contains an area that the agent is able to identify as the final area, after the robot reaches that area, it must return to the initial position as fast as possible. The maze contains vertical and horizontal walls that the agent should avoid collide with the help of some sensors and actuators. This paper covers the description of the agent, the software that was developed in order to overcome the challenge.**

***Keywords* – robotics, agent, ua, rmi, pathfinder, deliberative**

I. INTRODUCTION

This project was created for the course unit Intelligent and Mobile Robotics at the University of Aveiro. It consists in developing an autonomous agent, which is a system that tries to understand the environment using sensors and acts according with it using actuators (described in the section III), in this case, after it sense it will plan and only then actuate, a deliberative agent.

This agent will have to find its way to the final target through a maze. This maze contains walls that might be vertical or horizontal. After the robot finds the final target, it must return to the initial position in the less possible time. Besides the description of the problem, the strategies adopted to fulfil every challenge are described, e.g. strategy adopted for the agent to explore to maze till it finds the final target.

Every aspect of this paper was simulated using a specific platform and was never implemented in a non-simulated environment.

The agent described is able to find its way to the target and it is also able to return to the initial position after finding the shortest path, without any collisions.

II. SCENARIO

A. Description

This scenario is simple, it contains only walls and a final target that the robot is able to identify using the available sensors. The image 1 contains one of the possible mazes, all the maze is completely static and does not change over time.

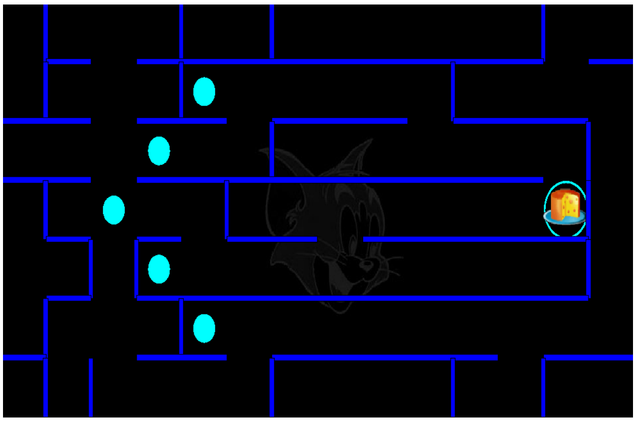

Figure 1
ONE OF THE POSSIBLE MAZES

In the image 1 it is possible to identify every aspect of a maze. It might only contain vertical or horizontal walls that can be identified with the dark blue color.

The ground is represented with black color and it is where the robot is able to move without any problems. There is also a cyan color represented in the figure, as a circle, the five cyan circles are the possible initial positions for the robot.

At last, there is a cheese on the map that is in the middle of a cyan circle, the agent must enter with all its body in the circle to be considered inside of the target area. The robot does not appear in the maze, but it is identified with a small mouse figure inside a circle and it is the only object that might change its position during the simulation.

B. Platform

The scenario is simulated using a platform, known as cibertools, available at `https://sourceforge.net/projects/cpss/files/cpss/`. Moreover, the version used is given with the agent's source code.

III. DESCRIPTION OF THE ROBOT'S SENSORS AND ACTUATORS

The following image represents the sensors and actuators of the robot used.

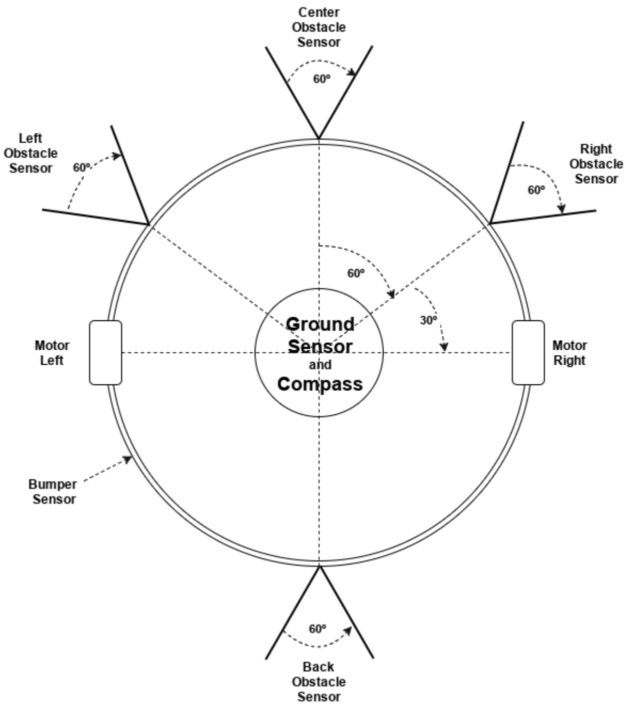

Figure 2
ROBOT SENSORS AND ACTUATORS

A. Actuators

Robot includes actuators that allows the robot to move in the world, in this case, two motors that control two wheels. These actuators do not have any kind of encoders, so it is not possible to odometry based on encoders. Although it is possible to estimate the robot pose using a movement model according with the velocity sent to the motors, which is described in section V.

B. Sensors

The robot has four sensors available: **GroundSensor, ObstacleSensors, Compass, Bumper and Time**.

GroundSensor allows the agent to detect when he has reached a target area by returning the ID of the area instead of -1 on the sensor.

ObstacleSensors are composed of a set of four obstacle sensors that allow to detect any walls that are in the way of the sensors. It was allowed to choose the position of each obstacle sensor, so it was decided to have one in the front of the robot (0 degrees). Since the robot moves most of the time in front, it is important to has such a sensor to detect front walls as soon as possible.
Two other sensors were placed 60 degrees of the left and the right of the center obstacle as shown in figure 2 for the robot to be able to mapping walls that are in the left and in the right. These sensors were not placed in 90 degrees because it would detect walls on the left and right a bit too late, and that walls are important when the robot is doing turns.
Another obstacle sensor was placed in the back of the robot to confirm the mapping that was performed by the other sensors.

The **Compass** sensor was used in order to obtain the robot orientation in the world, it is important for mapping, move to a specific position and estimation of the robot's pose.

There is also a **Bumper** that allows the robot to detect when it has collided with an object.

IV. ROBOT ARCHITECTURE

The robot is a **deliberative agent when returning and even when exploring**, it is always planning what is the next step to take.
There is 3 different main phases of the robot. It starts looking for unexplored zones until it finds the target area, which is the first phase. The second phase allows the agent to detect the fastest path to return to get the best time. At last, it must return to the start area.

This phases were represented with 7 different states: STOPPED, EXPLORING_OBJECTIVE, EXPLORING_FINAL_PATH, RETURN_TO_OBJECTIVE, PREPARE_TO_RETURN, RETURN_TO_START, FINISHED. These states are described in the section VII.

The image 3 represents in a general way what the agent does every cycle:

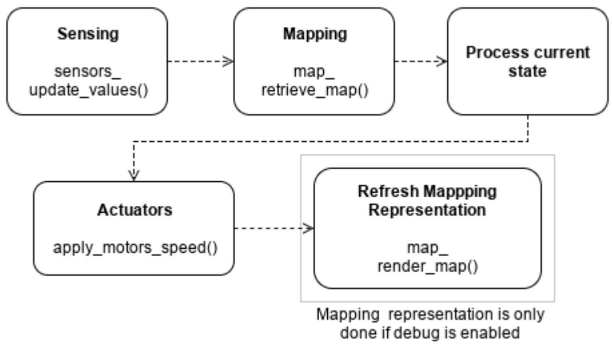

Figure 3
AGENT CYCLE FLOW

Besides the states, the agent is always sensing and actuating. When **sensing** there is some filters applied to the sensors. The source code of the agent easily allows to change to filter parameters used for every sensor, at the moment, the only sensor that has a filter is the obstacle sensor which contains a mean filter with 3 values to prevent major errors from the sensor.

After sensing, the robot verifies the values from the obstacle sensors and applies the proper mapping (section VI).

Then the agent will process the current state and apply the respective values to the motors.

At the end of every cycle, the agent refreshes the visual representation of the mapping done only when debug is enabled, otherwise, there is no visual representation for the user.

V. ESTIMATION OF THE ROBOT'S POSE

Since the robot uses his position when exploring to know if it was in area already visited, estimate the robot's pose was one of the first things that has been done.

The robot does not have any encoders associated to the wheels, then a movement model was been applied, but for that it must know the velocity output instead of the velocity that has been sent to the motors, it is not the same.

The following formula is the one that the simulator uses to generate the velocity output:

$$out_t = (in_t * 0.5 + out_{t-1} * 0.5) * noise$$

Formula takes half of the last speed applied and half of the requested speed and that will be the speed for the next cycle. It is impossible to know the noise (just possible to estimate) that has been applied to the motors, so it was ignored when estimating.

Now that the agent knows the velocity output, it can calculate his position using the following formula for his movement:

$$lin = \frac{out_{\text{right}} + out_{\text{left}}}{2}$$

$$x_t = x_{t-1} + lin * \cos(\theta_{t-1}) \quad y_t = y_{t-1} + lin * \sin(\theta_{t-1})$$

The following rotational formula was just ignored because θ accumulates the error of the previous θ, so it will have a big error after an amount of time.

$$rot = \frac{out_{\text{right}} - out_{\text{left}}}{robotDiam}$$

$$\theta_t = \theta_{t-1} + rot$$

Instead of estimating the robot's angle in the world, it was decided to use a sensor to do that, the compass. The compass has a gaussian error but it does not accumulate the error from the previous value, this was the reason to give preference for the compass instead of estimating the rotational using the movement model.

VI. MAPPING

To perform the mapping, the agent uses the obstacle sensors and the ground sensor. Mapping assumes that the movement model does not take into account error generated by the movement model, just takes into account the error from the sensors.

A. Structure to store the mapping

The structure used is a bi-dimensional vector which contains a wall counter, ground counter, visited flag and a state (might be WALL, GROUND or UNKNOWN).

Every time that a counter is incremented, the state is recalculated and it is the only value that is available for anyone that uses the Map, other values are internal.

Visited flag is used for the ground sensor, the agent is sure that it can move in that position if the robot is standing on it.

B. Strategy adopted to fill the mapping

The ground sensor helps to represent the current position as a point that the robot is sure that it is safe to move.

That does not help to map walls, just the ground. Then the agent also takes into consideration the obstacle sensors. The agent knows his position based on the movement model and the angle where the sensors are located.

To estimate where the walls are, the agent must calculate the sensors position and then estimate where the walls are, because the value returned by the sensors is related with the sensor position and not the center of the robot position.

To calculate the sensor position, the agent uses the following formula (R stands for robot):

$$sensor_x = R_x + cos(R_{compass} + \theta_{sensor}) * R_{radius}$$
$$sensor_y = R_y + sin(R_{compass} + \theta_{sensor}) * R_{radius}$$

With these two formula, the agent is able to calculate where the sensor is exactly in the world (instead of using the center). Every sensor is able to detect within a range of 60 degrees. In this case, the agent generates a line starting from the sensor in -30 degrees, 0 degrees and 30 degrees till the range detected.

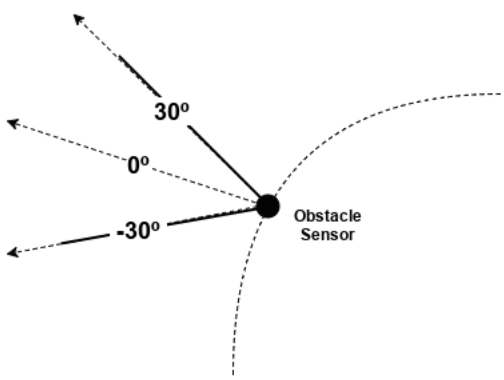

Figure 4

LINES THAT THE AGENT GENERATE IN ORDER TO CREATE THE MAPPING

Every reading that is bigger than 1 robot unit is discarded, this rule was created to prevent the obstacle noise. When the values are bigger, they're susceptible to more noise.

Then three lines with 100 points (according with figure 4) were generated till a 1 robot unit, if the value from the sensor is below 1 unit, the last value is marked as wall and the other values are marked as ground. If it is equal or bigger to 1, everything is marked as ground.

C. Decision rule to choose between GROUND / WALL / UNKNOWN

The easier state to define is UNKNOWN, which happens when the position was never changed or detected. If the wall counter has at least 0.15 times the ground counter, then it is marked as WALL, otherwise, it is GROUND.

```
map_[x][y].state = map_[x][y].wall_counter <=
    map_[x][y].ground_counter * 0.15 ? GROUND :
    WALL;
```

This value was several times changed during the development, and it was found as best value. It had to be a value to fulfil to opposite condition:

1. It could not block a way where it was possible for the robot to move.
2. It could not mark as possible a way where it was not possible for the robot to go.

These two aspects would totally affect the planning.

D. Visual representation of the mapping

At the end of each cycle (and if debug is enabled), the representation of the mapping can be visualised with a graphical window which was created using SDL2.

The image 5 is an example of the mapping:

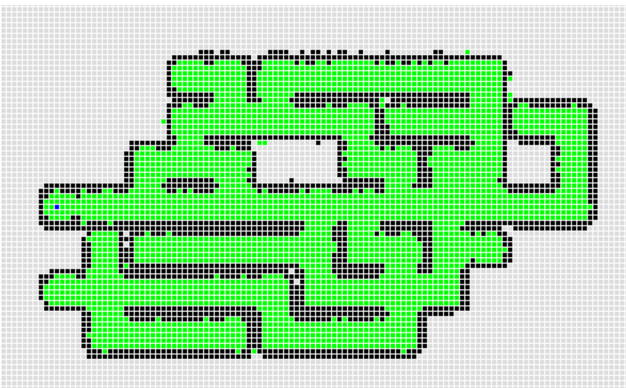

Figure 5
SIMPLE EXAMPLE OF THE MAPPING CREATED

The mapping also allows to represent the trajectory that was planned by the robot (even showing the best path to return and the current objective) and it can be found in section VII-E.

VII. AGENT'S BEHAVIOUR AND ITS STATES

The agent has the following states implemented to fulfil the objective.

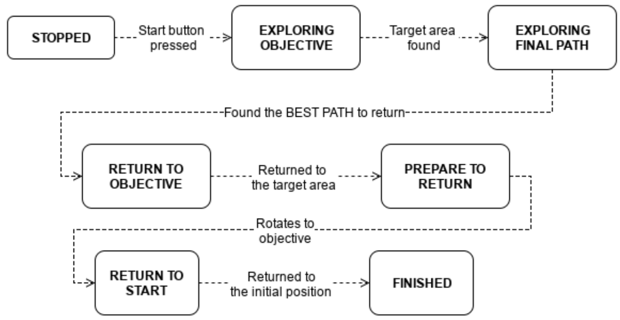

Figure 6
STATE FLOW OF THE AGENT

These states are described in the following sections.

A. Controller used by the agent

The majority of those states use a controller that follows a specific point. The controller takes into consideration the angle between two points, the current position of the robot and the objective for the next position. The error is the angle multiplied by a specific normalization factor.

```
// Error is the angle between the objective and
    the current position
error =
    angle_between_two_points(position_.get_tuple(),
    dst) + sensors_.get_compass());
error = normalize_angle(error) / NORM_FACTOR;

// Integral error calculation and feedback were
    omitted
correction = KP * error + KI * integral_error +
    KD * (error - last_error);

set_motors_speed(BASE_SPEED + correction,
    BASE_SPEED - correction);
```

B. Algorithms used for path calculations

B.1 A* (A-star)

This algorithm was implemented with a list of open nodes and closed nodes [1]. A node is a recursive element that might contain a reference to another node, in other words, it might represent a path.

The open nodes are the list of possible paths to explore. That list is sorted by cost plus heuristic. Where cost is the current cost until the current node and heuristic is the estimated cost from the current node until the final node.

Every time that the agent uses A* star, it uses the Manhattan distance. "*The Manhattan distance function computes the distance that would be travelled to get from one data point to the other if a grid-like path is followed*" [2]. This distance was used because when expanding of node, it was considered that the robot could only move in 4 ways, left, right, up or down. The robot does not move in diagonal.

B.2 Flood fill

This algorithm is really similar to a Dijkstra algorithm, it expands till it finds the final target. But it has this name because the difference from flood fill and Dijkstra is that it does not need a target objective to stop, while Dijkstra does need.

Flood fill used the same strategy as VII-B.1 but the heuristic is 0, then it behaves like a Dijkstra algorithm. The stop condition used was until it finds an UNKNOWN position. The specific usage of this algorithm can be found in section VII-D.

C. State STOPPED

This is the initial state of the agent. While in this state the agent does not waste any time and starts mapping with the available info. When the start button is pressed, the agent changes his state to EXPLORING_OBJECTIVE.

D. State EXPLORING_OBJECTIVE

The agent stays in this state until it finds the target area. Moreover, the robot needs to move in the maze in this state to find the target area.

For that objective, it has been implemented a **Dijkstra algorithm** to find the **nearest exit that is unknown** for the agent. The reason why it was picked Dijkstra instead of A star is that Dijkstra does not require a target to be used, this special case of Dijkstra might be also known as **Flood Fill**. It is exactly the same as A star, but it considers that the heuristic is 0.

The stop condition for the flood fill is not a coordinate in the map because it does not know the coordinate of the nearest UNKNOWN position, that what the algorithm is trying to find. So, the stop condition is when it finds a UNKNOWN position and that will make the **flood fill** return the next trajectory.

In the image 7 it is possible to check in red the trajectory that the robot wants to explore.

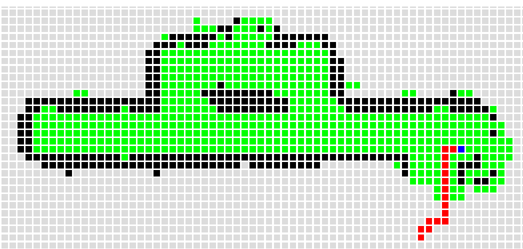

Figure 7

FLOOD FILL CALCULATING THE NEXT EXIT FOR THE ROBOT

At the first the algorithm, considers only ways that the diameter of the robot can go through, if there is no path available for that, it tries to find any way out using the same algorithm but with the minimal cell size.

After getting the path, it just uses the controller mentioned in section VII-A to follow the path. If it finds any obstacle in front, or the path gets empty, it just recalculates a new path.

E. State EXPLORING_FINAL_PATH

In this state, the agent already knows the position of the target area. So its objective is to know if the path discovered is already the best path.

This verification is done because the final time is only affected by the returning time, the time that the robot takes from the target area to the initial area.

To discover if it is the best path, it applies 2 algorithms:

1. A* algorithm where it can only be interconnected with nodes marked as GROUND (UNKNOWN positions are not valid for the robot to go in this case) and the initial position is the target area and the final position is the initial area - This path was named as known path.
2. A* algorithm where it can be interconnected with nodes marked as GROUND or UNKNOWN - This path was named as unknown path.

If the unknown path and the known path have the same size, it means that the agent already has the best path.

In the image 8 it is possible to verify both paths being represented. The orange path is the known path, interconnected using only KNOWN nodes and the purple path is the unknown path, it might be better than the known path, but there might be a wall in there.

It is possible to verify that the orange path is bigger that the purple path, although it might not be a possible path.

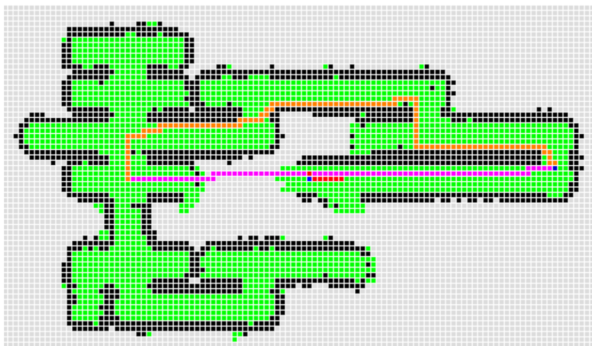

Figure 8

DISCOVERING THE BEST PATH TO RETURN BY UNCOVERING UNKNOWN POSITIONS

In the image mentioned before, the agent as the target area on the right and the initial area on the left and it does not have the best path yet, otherwise this state would be skipped.

F. State RETURN_TO_OBJECTIVE

In this state the agent already knows that it has discovered the best path to return, but it might not be in the target area, because it came from the state EXPLORING_FINAL_PATH that made it explore a bit more the map. So, now it must return to the target area.

To do that the agent applies an A* algorithm from the current position to the target area. After it reaches the target area, it changes its state to PREPARE_TO_RETURN.

G. State PREPARE_TO_RETURN

This state is simple, as the name says, the robot prepares to return. The agent knows that it is already in the target area, but it might not be best oriented to return and it does not want to lose time rotating while the time is already counting.

The agent rotates for the next position the it must go when returning, this will cut-off from the returning time the time that the robot would lose rotating for the next point. After it successfully rotates, it enables the ReturningLed and goes to the state RETURN_TO_START.

In more detail, to rotate for a specific point there is a controller available to do that, that the error is the difference from the angles between the two points. So it will be faster to rotate when the difference is still big and precise when the difference left to rotate is small.

H. State RETURN_TO_START

After the robot is correctly rotated for the next position and in the target area, it enters in this state.

In the image 9 it is possible to verify the path calculated to return.

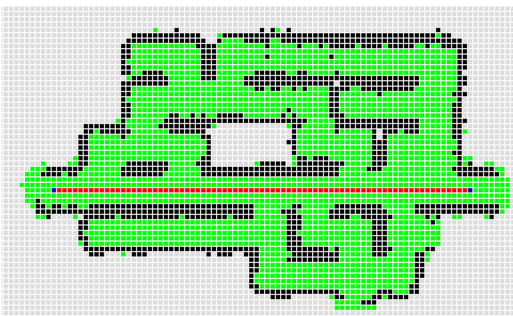

Figure 9
A* ALGORITHM REPRESENTED ON THE MAP TO RETURN

This state calculates a path between the target area and the initial area, using A*. And the agent moves until it finds that the distance with the initial area is close to 0.

I. State FINISHED

In this state, the agent simply informs the simulator that it has finished its operations.

VIII. BRIEF DESCRIPTION OF THE SOURCE CODE

The source code contains a CMakeLists.txt that is responsible to compile all the code and create the proper linkage with the necessary libraries. Beside the CMakeLists.txt, the source code contains the following files:

- RazerNaga.cpp/h - This file contains the rotation and moving towards a point controller. It also contains the flow of the state system.
- Map.cpp/h - These files are responsible to represent a map where which position contains 3 possible states: UNKNOWN, GROUND and WALL. This also contains interface for the RazerNaga to access the algorithms.
- MapAlgorithms.cpp/h - Responsible for the flood fill and A* algorithms.
- MapSDL2.cpp/h - Responsible for the visual representation of the map using SDL2.
- Sensors.cpp/h - This contains an easier interface to access values that were filtered.
- Filter.h - This file contains a mean filter implemented using templates.
- Position.cpp/h - This is responsible for represent the position of the robot and calculating it as well.

IX. CONCLUSION

The agent is considered robust to solve the challenge.

The velocity that the agent does the maze is considered pretty high, which is good to see that the agent manages to solve the scenario at such speed. It does never collide either.

The robot was tested in several different mazes, with different kind of difficulties and it solved all of them with a perfect score or almost perfect. It always manage its way to the target and to return with the best path.

REFERENCES

[1] Rajiv Eranki, *Pathfinding using A* (A-Star)*. MIT Edu. Accessed in 20 November 2016.

[2] Improved outcome software, *Manhattan distance definition*. Accessed in 21 November 2016.