AN AUTONOMOUS HYBRID ROBOT SYSTEM TO NAVIGATE THROUGH UNKNOWN MAZE ENVIRONMENTS

Pedro Ribeiro
Faculdade de Ciências & LIACC
Universidade do Porto,
Rua do Campo Alegre 1021/1051
4169-007 Porto, Portugal
email: pribeiro@dcc.fc.up.pt

ABSTRACT
This paper describes a fully complete autonomous hybrid robot system, named YAM (Yet Another Mouse), that is able to navigate through an unknown maze environment. YAM effectively tackles the problem of how to represent the environment using its sensor data to produce probability maps of the walls and beacons. Besides that, it is capable of computing long-term path plans using an adapted breadth-first search algorithm. It also shows how it is possible to model the actual motors behavior as a reactive task, using artificially created virtual short-term goals. We give real contest results, showing how YAM behaved on the "Ciber-Rato" robotics competition.

KEY WORDS
Intelligent Agents, Knowledge Representation, Path Planning, Robot Competitions.

1 Introduction

Since 1995 the University of Aveiro (from Portugal) has been hosting a competition among small autonomous mobile robots [1]. The competition is named "Micro-Rato" (Micro-Mouse) and it is organized by the Electronics, Telecommunications and Informatics Department. Although the rules have changed throughout the years, the basic problems tackled by the competition are localization and navigation. The robot has to move in an enclosed area (a maze) and needs to move itself to a goal area (a beacon).

In 2001, a software simulation of the "Micro-Rato" contest was initiated. The general idea was to provide very similar problems and rules but allow the participants to focus on more high level aspects, giving all of them an homogeneous virtual robot. This contest was named "Ciber-Rato" (Cyber-Mouse) [2]. As in its "brother" hardware version, "Ciber-Rato" (CR) has changed a bit along the years, but the basic set of sensors and actuators have remained the same.

We created a complete specification of a robot system, named YAM, capable of solving unknown maze environments, having the secondary goal of creating a successful entry to CR contest. The system is capable of transforming sensor data into a suitable symbolic probabilistic representation of the maze. It is also able to calculate near optimal paths between its goal points and it makes use of an hybrid architecture to reactively decide which real motor powers it should use.

We will start by making a brief description of the CR simulation system. Then, we will describe all components of YAM's system and architecture, seeing how it provided technical solutions and practical answers to the challenges presented. We will continue showing official results obtained in real contests and finalize making our conclusions and talking about some possible future work.

2 The "Ciber-Rato" Simulation

CR uses a simulated networked software simulation. The simulation has three basic components: the simulator (responsible for the actual implementation of the robot bodies, the maze and all sensors and actuators), the visualization system (which graphically depicts what is happening) and the robots themselves (which are made by the competitors). A single maze run can have at most three different robots. All robots (as well as the included visualization system) communicate with the simulator using predefined XML messages through UDP sockets. The simulation is not continuous, but discrete, in the sense that it is cycle driven. Each cycle the simulator gives to robots their sensor measures, waits for the actuator values, calculates all new positions and continues to another cycle. The cycle time varied during the years, and in 2006 it was between 40 and 80 ms. From now on, every time we specify some simulation parameter, we refer to the current competition values (2006 edition [3]), even if in the past that value was different.

A robot is simulated as having a circular form with its diameter representing a single basic unit on our virtual world. They have available a set of four obstacle sensors that measure the distance between the robot body and all obstacles, including walls and other robots. Each obstacle sensor covers a 60 degrees aperture, has a 0.1 resolution and it gives measures with some addictive gaussian noise, following a 0.25 standard deviation. One beacon sensor for each beacon is also available, with a 120 degrees aperture, a unitary resolution and as before this sensor accumulates a gaussian noise with standard deviation 5. There exist bigger walls that hide the beacons to these sensors. The robot also has a compass sensor that measures the orientation of the robot with the same noise and resolution of the beacon sensor. Finally, all agents have a collision detector (a bumper, to tell if the robot has collided with a wall or another robot) and an exit detector (that tells if the robot is inside a goal area). To actuate, two side motors are available, which we can activate in a $[-0.15, 0.15]$ power range, with a resolution of 0.001. As before, motors have gaussian noise, in this case with a 3% standard deviation. Besides that, robots have different LEDs available that are used to indicate other events such as the end of its movements.

The maximum size of a maze is 14×26. The goal of a single CR run is to visit a set of different beacon areas, signing the arrivals with a LED, and then go back to the starting position as fast as possible, signing another LED. To achieve this, the robots have a limited time (between 1800 and 3600 cycles). Penalizations occur for each collision we provoke, and bonus points are given for each beacon successfully visited. The time the robot takes is only measured in the trip between the last beacon visited (signaled by another specific LED) and the start point. The simulator calculates the best time that could be achieved in this trip and penalizes the robot if it spends more time than the best expected time.

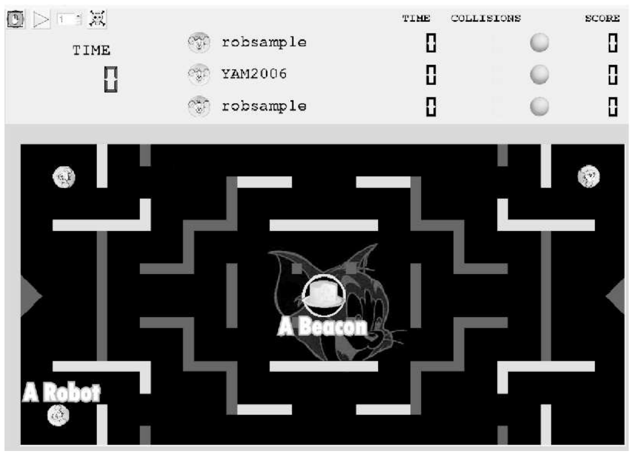

Figure 1. An example maze seen on the default contest visualization system.

An example maze can be seen on Fig. 1, showing how the default CR visualization system represents it. For a more detailed description of each year's competition rules, check [4].

3 YAM Description

In this section, we describe the architecture of YAM showing how it reasons and acts.

3.1 Architecture

YAM aims to produce good decisions within each execution cycle. We used a simple single threaded architecture as it proved capable to produce decisions quite fast, not spending more than the available time for each cycle. This architecture is described on Fig. 2.

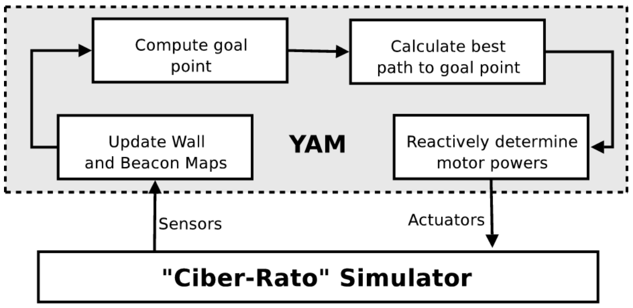
Compute goal point
Calculate best path to goal point
Update Wall and Beacon Maps
YAM
Reactively determine motor powers
Sensors
Actuators
"Ciber-Rato" Simulator

Figure 2. YAM Architecture.

YAM is an agent that uses an hybrid architecture [5], given that it mixes deliberative components with reactive ones. YAM stores a symbolic description of the world described in section 3.2 and reasons about it for finding the best path to a determined goal point as described in section 3.3. In this way, we can say that YAM is deliberative. However, when it gets to determine the actual powers it should give to the motors, YAM uses a “virtual beacon” to decide its next motor powers in a fully reactive way, without any knowledge of how the virtual beacon was found, as described in section 3.4.

It would be very difficult to debug YAM only with textual information, and therefore since the beginning a graphical debugger (“Yam-Viewer”) was created. This tool proved to be very useful and gave a way to verify the theoretical ideas. The debugger has several view modes and can be turned on or off on demand.

3.2 Inner World Representation

One of the most challenging aspects of this contest is how to internally represent a map of the maze. This is still a research problem in robot mapping [6]. One key aspect that almost all map representations have in common is the fact they are probabilistic [6]. YAM also follows this rule of thumb as we will see. Another important aspect is that the environment of the maze is mainly static (the walls). Given this, in order to efficiently use all the data that the sensors receive, YAM constructs an internal representation of its environment, divided in three components:

- Present Robot State (position and orientation)
- Wall Probability Map (where are walls located)
- Beacon Probability Map (where are beacons located)

Initially, YAM is considered to be in position $(0, 0)$, which means that we use the start point as the reference point of our coordinate system. We now describe in more detail each of the three components.

3.2.1 Present Robot State

In each cycle YAM uses its compass sensor to determine its orientation. This value, although it has some noise, is always a good approximation of the real value. In contrast, we could infer the orientation based on the powers given to the motors, but that would accumulate all noise as time goes by, and in a practical experience it proved to be better to use the available compass.

Knowing our orientation, YAM calculates its new position in the map, given as a coordinate (X, Y), using the rule described in equations 1, 2 and 3. In relation to cycle n, $\mathbf{X}[n]$ and $\mathbf{Y}[n]$ represent the agent position coordinates, $\mathbf{L}[n]$ and $\mathbf{R}[n]$ represent the powers given respectively to the left and right motors and $\theta[n]$ represents the robot orientation.

$$\begin{aligned} \mathbf{Linear}[n] &= (\mathbf{L}[n] + \mathbf{R}[n])/2.0 && (1)\\ \mathbf{X}[n+1] &= \mathbf{X}[n] + cos(\theta[n]) \times \mathbf{Linear}[n] && (2)\\ \mathbf{Y}[n+1] &= \mathbf{Y}[n] - sin(\theta[n]) \times \mathbf{Linear}[n] && (3) \end{aligned}$$

Although some error is always introduced by the motor and compass noise, this proved to show no real significative deviation to the real robot position and orientation. This is very important, since eventual odometry errors on this phase could be a cause of big problems.

3.2.2 Wall Probability Map

The maze has a 14×26 maximum dimension. To facilitate its representation, the maze was discretized and transformed into a grid of 140×260, meaning that we used a granularity of 0.1 units. Since the origin of our reference system is the start point, and the robot can start in any position of the map, we have to reserve memory to four times the maze maximum dimension, in order to analyze the robot's movement in any direction. This results in a structure of 280×520 being kept in memory.

Each point of the grid keeps a floating point value, representing the probability of a point having a wall. The probability is saved in a scaled way, in the sense that it can be positive and negative. Initially, each point has probability zero. Then, on each cycle, several updates are made. First, the points occupied by the agent body are marked with minimum probability, since if the robot is on that position, no obstacles exist there. After that, based on the values received by the obstacle sensors, the wall map is changed, as depicted on Fig. 3.

If a sensor indicates a value V, it means that an obstacle (an other mouse or a wall) exists approximately at the distance $1/V$, somewhere in the sensor range.

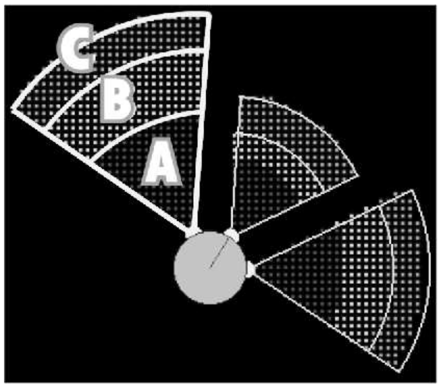

Figure 3. YAM's use of obstacle sensors (from YamViewer).

This value V has a maximum noise, given by the constant $noiseObstacle$. All areas closer than $1/(V + noiseObstacle)$ (the worst case) are marked with minimum probability (area A in Fig. 3). Then, until distance $1/V$ (area B in Fig. 3), the probability of having wall in this area is reduced by a value proportional to the distance. Finally, from $1/V$ to $1/V + minWallWidth$ (area C in Fig. 3, where $minWallWidth$ represents the minimum wall width, available at the beginning of the simulation), the map is updated, with the probability of having wall increased in proportion to the distance. This behavior can be summarized mathematically with the equations 4 to 10, where d means distance, and p_{xy} indicates the probability of a single point having a wall. $dFar$ is a custom defined constant, and $inc(d)$ is a value proportional to the point distance, indicating how much should we change the wall probability.

$$\begin{aligned} d_A &= 1/(V + noiseObstacle) && (4)\\ d_B &= 1/V && (5)\\ d_C &= 1/V + minWallWidth && (6)\\ inc(d) &= round(dFar/d \times 4) && (7)\\ dFar \le d_{xy} \le d_A &\rightarrow p_{xy} = minProb && (8)\\ d_A < d_{xy} \le d_B &\rightarrow p_{xy} = p_{xy} - 4 \times inc(d_{xy}) && (9)\\ d_B < d_{xy} \le d_C &\rightarrow p_{xy} = p_{xy} + inc(d_{xy}) && (10) \end{aligned}$$

YAM ignores sensor values when $1/V > dFar$, because when V get smaller, the noise is so predominant that the values become unusable. In the current YAM implementation, an empirically verified value of 25 grid units is used for $dFar$. Also, note that a point takes more time to become marked as a wall (as enforced by the factor four in equation 9) to represent the fact that the sensors don't differentiate between an obstacle occupying a small part of the range and another one occupying all the range. In this way, it is not easy for a wall to "appear" in the map and more quickly false walls are unmarked. Finally, the formula for $inc(d)$ (equation 7) was obtained using empirical tests with different models. It is important to note that it grows bigger as the distance gets smaller, that is, when we have more certainty, as seen on Fig. 4.

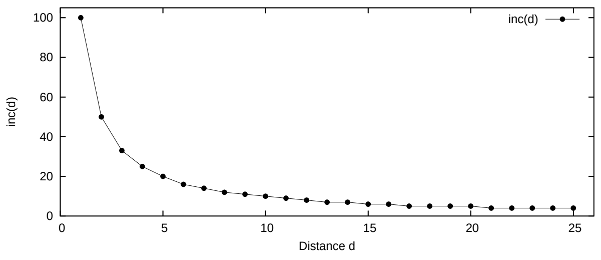

Figure 4. Value of $inc(d)$ in relation to point distance d.

Figure 5 illustrates how YAM successfully constructs an internal representation of a real map.

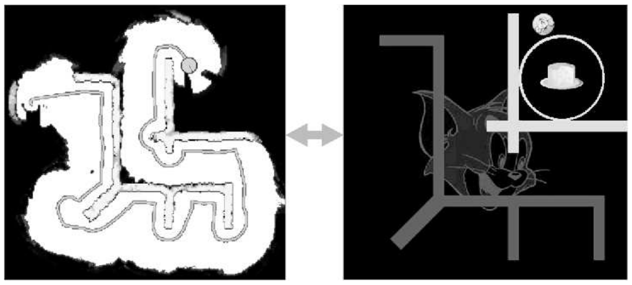

Figure 5. Comparison between a YAM wall probability map (from Yam-Viewer) and the real map used.

3.2.3 Beacon Probability Map

In a similar way to the wall probability map, each beacon (a goal area) has a probability grid map. Each point is marked with the probability of the respective beacon being there. Let $rel(P)$ be the relative angle of a point P to the value measured by beacon sensor (which indicates that the beacon should approximately be at that angle). The probability of the point P, represented by $probBeacon_P$ is updated as described in equation 11. In this way, the smaller the value of $rel(P)$, the bigger the probability of it being the real beacon.

$$probBeacon_P = probBeacon_P + 50 - rel(P) \qquad (11)$$

The best estimate of each beacon is the point with higher probability. In order to maintain the values of probability manageable, all probabilities are decreased by a constant value each time the maximum probability point reaches a defined threshold. With this apparently simple

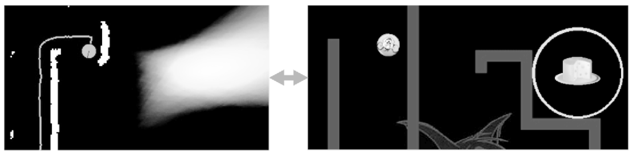

Figure 6. Comparison between a YAM beacon probability map (from Yam-Viewer) and the real map used.

probability map, very accurate beacon maps can be created, as the one of Fig. 6.

Sometimes big walls may be completely hiding all beacons, and in this case no single prediction of the beacons positions can be made. This would happen no matter what method was used. Our method always provides a good estimate, based on all the information the beacon sensors have given. During different editions, the beacon sensor has suffered some changes (big walls did not exist, the sensor had a bigger range and now the sensor has some latency), but the initial implementation has always provided good results and has not been modified.

3.2.4 Precalculation

Since the tasks described in the last two sections can be very time consuming, virtual “masks” containing all the possible updates for different angles (for walls and beacons) are precalculated in the beginning. When moving on a map, YAM simply puts that masks above its actual position and updates its values in consonance.

3.3 Path Planning

The goal of the contest is to reach all beacons, and then choose one of those beacons to initiate a return trip to the starting point, as fast as possible. It should be noted that no globally perfect strategies can be followed and real behaviors are always somehow dependent on the secret maps chosen. YAM aims to be globally good and present a coherent performance in all maps.

If there are no estimates for the positions of the unvisited beacons, YAM starts to traverse the maze as a whole. It looks for unexplored map areas on the edges and gradually goes to inner areas (as the map is being explored), following a somehow spiral path. Using this strategy, and if time allows, it is guaranteed that YAM traverses the entire maze. Edge areas have priority as they normally provide a better range of vision for discovering existing beacons.

Once YAM estimates the position of one or more beacons, it chooses the closest unvisited beacon and marks it has its goal. After visiting all beacons, YAM chooses the one closer to the starting point as the one he will choose to activate the return LED and then make the return trip. Remember that after this LED is activated, the clock ticks and the robot is penalized for any extra time it consumes in relation to the best possible path. Before using the LED, YAM also sees if it still has time to search unexplored areas to verify if they can provide quicker paths than the ones already known. Once it thinks it is no longer safe to search for unknown areas (the time is running out) or simply it has already all the information it needs, YAM chooses the starting point as its goal.

No matter what is calculated, YAM always chooses a goal point as its objective, being it an actual real beacon, the starting point, or a virtual beacon (when it traverses the maze in a spiral way). Then it calculates a path from the current robot position to the goal point. Several methods for this exist [7], but almost all of them, like the A* algorithm, present the restriction of not always finding an optimal path. Instead, they calculate an approximation, using some heuristics to trade accuracy for time.

In YAM's case, the cycle time proved to be more than sufficient to use a complete and adapted breadth-first search. The search starts by putting the goal point on a list. Then, the first point of this list is taken until it corresponds to a point belonging to the robot body. Each time a point is taken from the list, its neighbors are added to the end of the list. And it is the definition of neighbors that YAM uses that makes it able to calculate the path in an affordable way. In fact, the neighbors to a single point correspond to a jump of size N, first in each vertical direction, and then on each diagonal direction. Each point has therefore eight neighbors. Augmenting N, we shrink the actual map being scanned. A value of N too big would cause the search to "jump" walls, giving a wrong path, since it would imply that the robot could move through the walls. The current implementation of YAM uses $N = 3$.

YAM considers all points above a defined threshold as being a wall and all others as not having wall. It also calculates if the center of the robot body could be situated on that point, guaranteeing that no body points would touch walls. Finally, it should be noted that when returning to the starting point, YAM only uses already explored points of the map, avoiding unexpected surprises that could cause penalizations. Remember that the time it takes to return to the starting point is very important in the final score. Figure 7 illustrates a path to a goal point.

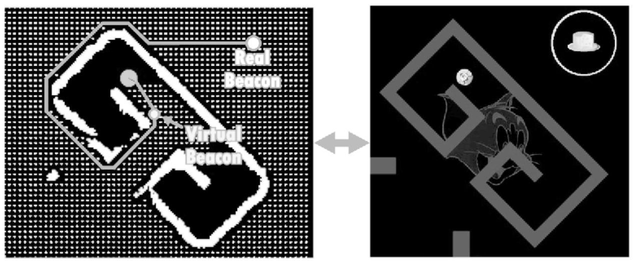

Figure 7. Comparison between a YAM generated best path and virtual beacon (from Yam-Viewer) and the real map used.

3.4 Motor Powers

With the best path to the goal point already planned, YAM still has to decide what specific powers it should give to its motors to traverse that path. This is done by creating a "virtual beacon" in the more distant point of the path that YAM can still see with no obstacles in the middle. This is done using Bresenham line algorithm [8] to draw virtual lines to every path point and checking if they touch any obstacle. Figure 7 exemplifies a virtual beacon.

Having the virtual beacon for the current cycle, the robot decides how to act with a completely reactive algorithm, manually tuned to make the robot avoid obstacles and move parallel to walls (to avoid bouncing its body). Figure 8 briefly describes the algorithm used. YAM has obstacle sensors on its body radius, with two sensors on the center and two side sensors which are located on a 60 degrees angle distance to the center. $lightDir$ is the relative angle to the virtual beacon and $tooClose$, $dirTooFar$, $dirTurn$ and $smallClose$ are manually tuned constants. When the robot is close to an obstacle or to the final starting point, its speed is also decreased to achieve better precision.

```
IF (anySensor < tooClose OR Collision)
   ROTATE_BEST_SIDE
ELSE IF (abs(lightDir) > dirTooFar)
   ROTATE_RESPECTIVE_SIDE
ELSE IF (abs(lightdir) > dirTurn AND
         respectiveSensor > smallClose)
   MOVEANDROTATE_RESPECTIVE_SIDE
ELSE
   FULL_SPEED_AHEAD
```

Figure 8. YAM's reactive pseudo-algorithm for following a virtual beacon.

4 Competition Results

Since 2002 YAM participates with its basic architecture and algorithms intact. As said before, several rule changes were made, but YAM behavior was created having some of that possible changes already in consideration. Table 1 shows how did YAM ranked on all the contest editions it participated in. All competition editions were made having two start rounds that selected the nine best robots for the third round, which then selected the three best robots to participate in a final round. This last round decides the contest for itself, not accounting the robot's score on previous rounds.

As we can see, YAM quickly became the champion and obtained the first place in two consecutive years. In 2004, for the first time, YAM did not reach the final. On the first two rounds of that year, YAM obtained by far the best

Edition	2002	2003	2004	2005	2006
YAM Rank	1st	1st	9th	2nd	4th
Nr Participants	13	14	13	10	7

Table 1. YAM global rank on Ciber-Rato competitions

overall results. On round three something really strange happened and our agent simply started to rotate on the same place. After the end of the competition, the organization did some log checking and saw that some network problem occurred and informally told me that. No complaint was formally made, respecting the competition spirit, but it was still to be proved that YAM was not the best robot present in the competition. In 2005, for the first time, our robot behaved normally and another robot ("El Raton") had a better behavior. YAM reached the final, but the winner was the only robot which explored the fact that a returning led was introduced, continuing to explore the map for better paths while there was time (as now YAM does), gaining a decisive advantage. This was a risky move, but one that has proved to be right. Finally, in 2006, YAM did not reach the final, having the same points has the 3rd place robot, but losing to it in a special tie-break rule, which was used for the first time, to decide the access to the final.

More detailed results and logs of the contests can be obtained in [4].

5 Conclusions and Future Work

YAM is a fully functional autonomous robot, using an hybrid agent architecture. It can localize itself and navigate trough a maze-like environment. YAM successfully maps sensor measures to a probabilistic map representation of the maze. It calculates near optimal paths to achieve long-term plans and has a reactive behavior to follow those paths, using "virtual beacons". Also there was no single maze used on a competition that YAM could not solve in time.

Although the competition rules have changed a lot during the years, the initial design of the agent is almost intact, which proves YAM uses a robust and flexible architecture. More than that, YAM was initially the undisputed competition champion and still remains very competitive. In our opinion, YAM can therefore be called a winning solution for maze-like virtual robot competitions and the experience acquired on its development and implementation can be useful to others.

One main limitation of YAM architecture is the impossibility to act on a more continuous world, since it has a completely cycle driven control flow. We plan to adapt and transform YAM to use a multi-threaded architecture, capable of reasoning almost in real-time. Other future planned work include a better and scientific analysis of the probability formulas for the maps, an odometry corrector for spatial misalignment provoked by long runs and a study of different path planning algorithms, including iterative ones, that do not need to re-calculate the path in each cycle. One other area that we wish to explore is cooperation with the hardware version of the contest, helping in the high-level part of real robot competitors.

Finally, we plan to participate on upcoming CR competitions, trying to reacquire the first place and always thriving for a better competition and agent intelligence. Future rule modifications will certainly be a challenge to the so far adequate architecture, since the organization always try to make things more interesting and create new challenges to the competitors.

References

[1] L. Almeida, J. L. Azevedo, B. Cunha, P. Fonseca, N. Lau, and A. Pereira. Micro-rato robotics contest: Technical problems and solutions. In *Controlo'2006: Seventh Portugese Conference on Automatic Control*, 2006.

[2] Nuno Lau, Artur Pereira, Andreia Melo, António Neves, and Joao Figueiredo. Ciber-rato: Um ambiente de simulação de robots móveis e autónomos. *Revista do DETUA*, 3(7):647–650, September 2002.

[3] Regras e especificações técnicas da modalidade ciber-rato - edição 2006 da competição. http://microrato.ua.pt/main/Docs/RegrasCiberRato2006.pdf.

[4] Micro-rato contest. http://microrato.ua.pt/.

[5] M. Woolridge and N. R. Jennings. Agent theories, architectures and languages: A survey. In *Intelligent Agents, Proceedings of the ECAI-94 Workshop on Agent Theories, Architectures, and Languages*, 1994.

[6] S. Thrun. Robotic mapping: A survey. In G. Lakemeyer and B. Nebel, editors, *Exploring Artificial Intelligence in the New Millenium*. Morgan Kaufmann, 2002.

[7] Marten Klencke. Autonomous navigation in an unknown environment, 2005.

[8] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965.