

The Autonomous Agent Nai in RTSS 2007

Jorge Pinto

jpinto@cs.nmsu.edu

Computer Science Department, New Mexico State University, Las Cruces, NM 88003, USA

Abstract—This paper introduces the overall design of the agent Nai being developed for the RTSS 2007 event. Solutions and ideas for the major problems on implementing a competitive agent are introduced. A promising agent is presented by making use of learning techniques on key areas to maximize intelligent behavior under predictable and non predictable circumstances.

I. INTRODUCTION

The CiberMouse [1] competition, held at the Real-Time Systems Symposium (RTSS) 2007 conference, is a simulated environment for autonomous virtual robots with the following objective: explore the unknown to find a goal, and return to the starting point.

The agent Nai has participated in past editions of the Portuguese version of the event held at the University of Aveiro. It got its best marks in 2003 and 2004, with a 2nd place in both years.

Every year, a new detail is added or changed in the rules [1] of the competition to make it different, which implies that the participants have to update their agents. Examples of this evolution are for instance higher levels of noise introduced in the sensors, and delays in receiving the values read. This adds the challenge of being able to keep a valid internal representation of the world while trying to get back to the starting point.

II. ARCHITECTURE

The platform used for the development of the agent is Linux. The visualization approach for debugging is identical to the one used by agent YAM [2], where the XLib library is used to draw the internal state of the world in every cycle, allowing a real time visualization of what the agent is “thinking”.

In this version, SQLite3 is introduced as a database backend. Although still in a phase of development, the evaluation of the motors values to be used are learned from previous states, and kept for later improvement.

III. THE AGENT

There are many details about constructing a full capable autonomous agent for this competition. Here, the main design is introduced without going into much detail, but still revealing the core ideas. The agent is divided into the following areas:

- Read Sensors
- Update World
- Compute an Objective
- Find Path
- Calculate Motors

The following sections will follow the itemized points.

IV. READ SENSORS

The sensors are read every cycle, promoting changes in the internal representation of the world due to its input.

Currently, the agent implements a simple strategy of alternating the sensors to be read. There is no specific policy to take advantage of certain situations like if rotating on the same place and not making linear movement, then certain sensors could be read since there is no danger of hitting an obstacle.

Before going to the next step of updating the world, if the compass sensor was read, a tentative is made to correct the position of the Agent. Simply correcting the compass is not enough, so a history of the motor values applied is kept to later on adjust the agent position. The motors values from the corresponding reading cycle are adjusted such that the compass reading is achieved. From there, the stored motor values are reapplied and the new position is computed.

V. UPDATE WORLD

The internal representation is similar to many other contestants. The agent is positioned initially in the center of a space that is 4 times the size of the possible map. Updating the world takes place one time per cycle. When new sensor readings values are available, the agent “marks” the sensor values in its internal representation of the world. Each point marked as wall will have a surrounding with a minimal radius (the minimal wall width) where the *Find Path* computation will not use those points.

VI. COMPUTE AN OBJECTIVE

The competition has two main objectives: 1) find the goal (beacon); 2) return to the starting point.

The agent is equipped with a beacon sensor that allows to detect the beacon direction relative to the compass, with the following constraints: 1) the obstacles between the agent and the beacon have to be low obstacles; 2) the agent has to be turned to the beacon in a maximum difference of 60 degrees between the compass and the beacon.

Initially, the internal world is populated with fake beacons while the real beacon is not visible. The strategy here is to try to get to the nearest fake beacon. Once reached, it tries to get to the next, until all the map has been covered. When the real beacon gets visible, the possible area is marked, and the agent will try to cover that area search for the beacon. Eventually, it will start receiving the beacon signal and reach its goal.

VII. FIND PATH

To find a path between a previously computed objective and the agent, the A-Star (A*) algorithm is used.

The following function represents the cost between the source point and the target point, that passes through node n :

$$f(n) = g(n) + h(n)$$

Here $g(n)$ is the real cost from the source to node n , and $h(n)$ is the estimated cost between node n and the target. $f(n)$ is the total cost of the path that passes through node n . The heuristic function used is the computed grid distance, instead of the direct distance between two points.

$$h(n) = xdiff * step + ydiff * diagonal_step$$

Where $step$ is the grid step used to search for a path in the world, and $xdiff$ and $ydiff$ are the respective minimal distances to be done in x and y to achieve the target node. $xdiff$ and $ydiff$ are swaped as necessary to do the correct computation.

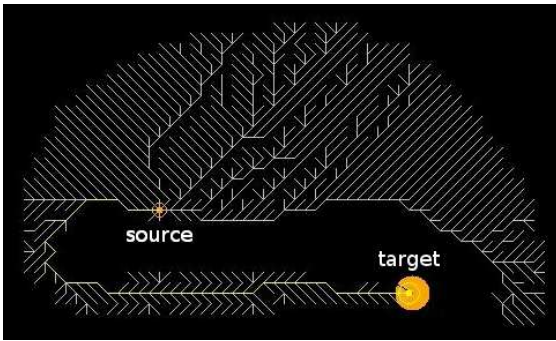


Fig. 1. Points visited around a wall, with lines to parent nodes in A*. The path is searched from the source to the target.

As seen on Figure 1, the strong point in using A* is the reduced number of points visited when compared with a simple bread-first search. What balances the decision between these two approaches is if the priority queue and the heuristic function used in A* are fast enough when computing big areas. If the same number of points are going to be visited, which might be the case when a path is not found, then bread-first search is better. This is important because a cycle where a computation takes too long might make the agent miss the cycle time to return an action.

Other approaches have been implemented to dynamically change the step value while searching for a path to make it more refined and to pass between tighter obstacles, but are not in use in this version.

VIII. CALCULATE MOTORS

One of the key points of the autonomous agents participating in this competition is the computation of the motors values.

A complex agent with good path finding and internal representation but unable to avoid obstacles in the fastest

way possible, is not very efficient. Most of the contestants, including previous versions of the agent Nai, have hardcoded values of how the agent should behave when the direction to follow is between certain values and the sensors do not put that action in danger.

The agent Nai solves this problem using Reinforcement Learning – Q-Learning. The main points of this approach involve a state representation, a set of possible actions to try with each state, and a reward function. The idea is to have the agent to try several actions on a seen state. Once it does something good, like reaching a desired point, a reward is given. From here, other actions and states will have this reward propagated (learned), which will cause the agent to start to use the actions with higher rewards to reach its goal faster.

The data structure used as the state, which is one of the keys, is the following:

```
State {
    bumper,
    left_motor, right_motor,
    beacon_distance, beacon_angle,
    obstacle_sensors[n]
}
```

To reduce the number of possible states, the values are computed to the nearest values. For example, using values $[0, 5, 10]$ and receiving a *beacon_distance* of 6 or 7 will keep in the state the same value of 5, while 8 and 9 will keep the value 10.

To prevent these states mapped to the actions from being lost, a database backend is being used. SQLite3 has available the source code and compiles easily on most platforms. Mainly, three tables are used: 1) index the state; 2) index the possible actions; 3) map a state to an action, with the current calculated Q value and reward.

Although still under development, primary tests reveal that this approach is viable, and that there will be cycle time left for the other necessary computations.

IX. CONCLUSION

This version of the agent Nai is still under development. There are no results other than a few tests and the theoretical view that this approach should work well and make the agent very regular in this and upcoming editions of the competition.

The day of the event is awaited between code development and tests.

REFERENCES

- [1] Departamento de Electrónica, Telecomunicações e Informática. *Ciber-Mouse: Rules and Technical Specifications*. Universidade de Aveiro, July 2007.
- [2] Pedro Ribeiro. *YAM (Yet Another Mouse) - Um Robot Virtual com Planeamento de Caminho a Longo Prazo*. (Revista do DETUA, Vol.1, N.6, September 1996), 2002