CiberMouse @ RTSS 2007

http://www.ieeta.pt/~lau/web_ciberRTSS07/

December 3, 2007

Autonomous Rescue Agents in the Lisbon-Dakar Rally

Team Reports

A satellite event of

RTSS 2007
The 28th IEEE Real-Time Systems Symposium
December 4-6, 2007
Tucson - Arizona, USA

Organizers

Luis Almeida, University of Aveiro, Portugal
Nuno Lau, University of Aveiro, Portugal
Artur Pereira, University of Aveiro, Portugal
Paulo Pedreiras, University of Aveiro, Portugal
Andreia Melo, Critical Software, Portugal

Jury

Giorgio Buttazzo, SSSUP, Italy
Steve Goddard, UNL, USA
Thomas Nolte, MRTC, Sweden

Teams that submitted the progress report

Team Name	Members	Institution
Roadrunner	Joshua Curtis Salvador Rodriguez Adam Critchley	University of Texas at San Antonio, USA
EvoRobert	Ali Alanjawi, Frank Liberato	University of Pittsburgh, USA
NAI	Jorge Pinto	New Mexico State University, USA
FAUBOT	Daniel Danner Caroline Kaufhold Peter Kranz Rainer Müller Sven Pfaller Christian Rieß Elli Angelopoulou	University of Erlangen-Nuremberg, Germany
RoboCoog	Ashish Kapadia Arun Chhetri Ronak Shah Albert Cheng	University of Houston, USA
TAMOUSE II - Swift	Jaya S. S. Palli Krishna Konda	Texas A&M University, USA

Sponsor

Support

Message from the organizers

The CiberMouse@RTSS2007 competition is the third in a sequence of similar events, following CiberMouse@RTSS2006 in Rio de Janeiro and The MARS Task in Miami in 2005. These events are organized as satellite events of RTSS with the purpose of giving (under)graduate students the chance to apply their research/graduation work on a problem that has many similarities with a true real-time embedded system and comparing their approaches by means of a competition.

This year we have 6 teams that made all the way up to writing the reports delivered in this booklet. These reports will be updated by the teams after the event and published on-line, on the event website:

http://www.ieeta.pt/~lau/web_ciberRTSS07/

Being technically similar to the previous edition, we expect a general improvement in the performance of the teams. Worthy of note is the case of TAMU, the Texas A&M University, which organized a local competition to select their participating team.

In terms of scenario, this year the contest develops within the scope of the Lisbon-Dakar Rally, which crosses the Sahara desert in north Africa. Three rally teams get lost in the desert during a violent sand storm. The backup teams send robotic rescue agents to rescue their cars and continue the rally. The cars send a localization RF beacon that is used to guide the rescue agents. No GPS info is available. Each agent can only rescue its own team. It is a race against time to rescue the car and continue the rally asap.

The challenge proposed to the students is to program the rescue agent to reach the RF beacon and return. In the way there are obstacles that may block the beacon and may even be dynamic since the agents run three at a time, frequently getting in the way of each other. An adequate robot motion control requires proper reactive real-time programming.

Finally, a jury, to which we would like to express gratitude for accepting participating in this endeavor, namely Giorgio Buttazzo, Steve Goddard and Thomas Nolte, will help in evaluating the teams performance, considering the competition results, the submitted report and the oral presentation on site. We believe that we have gathered all the ingredients for a stimulating and challenging event. A last word of appreciation to our sponsor, Critical Software, and to the University of Aveiro, IEETA and ARTIST2 for the support that allowed our own participation.

That all participants enjoy from this experience. Let the show begin!

Luis Almeida

Nuno Lau

Artur Pereira

Paulo Pedreiras

Andreia Melo

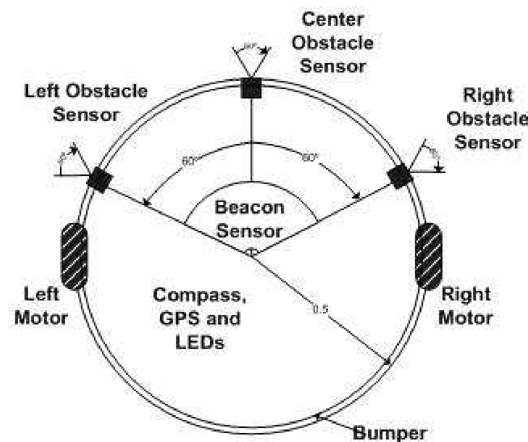
Technical challenge

Technically speaking, the task of the robots is to go from their starting position to the target area signaled by an active beacon, and then return to their starting position. The final score depends on the distance from the starting to the ending point, on the time taken and on possible penalties due mainly to collisions with obstacles and other robots.

A graphical front-end shows the operations area, the movement of the robots inside the area, their scores and provides a control panel to manage the competition.

The area is rectangular, delimited and populated with obstacles. Start point, target area and obstacles are unknown to the robots at the beginning. Moreover, there are obstacles higher and lower than the target beacon.

The virtual body of the robots has a cylindrical shape and is equipped with sensors and actuators. The sensory system is composed of obstacle sensors, target beacon sensor, compass, and a bumper. The main actuators are two diametrically opposed motors controlling two independent wheels in a differential drive fashion.



To motivate using real-time design methods, selective delays have been added in the access to the sensory information as well as limitations to the amount of sensory information that can be obtained with each simulator query. These features will require a judicious control of speed and selection of sensor queries in order to avoid bumping against obstacles and other robots which cause penalties. This year, in particular, the noise level added to the sensor readings has been reduced with respect to the last edition.

Finally, more related technical information can be found on the web sites of both the 2006 edition (http://www.ieeta.pt/~lau/web_ciberRTSS/) and the original competition behind CiberMouse, which is the Ciber-Rato / Micro-Rato competition carried out in the University of Aveiro since 1995 (http://microrato.ua.pt). Note, however, that there are several technical differences with respect to the latter ones.

Team Reports

Roadrunner 9

A. Critchley, J. Curtis, S. Rodriguez

EvoRobert System Description 13

Ali Alanjawi, Frank Liberato

The Autonomous Agent Nai in RTSS 2007 17

Jorge Pinto

FAUBOT: Purposeful Navigation of a Robot in a Simulated Environment.. 19

D. Danner, C. Kaufhold, P. Kranz, R. Muller, S. Pfaller, C. Rieß, E. Angelopoulou

RoboCoog: The CiberMouse Agent 23

Ashish Kapadia, Arun Chhetri, Ronak Shah, Albert Cheng

SWIFT – A Scenario based dynamic sensor scheduling robot .. 27

Jaya S. S. Palli, Krishna Konda

Roadrunner

A. Critchley, J. Curtis and S. Rodriguez

Dept. of Computer Science, University of Texas San Antonio, San Antonio, TX 78249

Abstract

The purpose of this paper is to describe some of the fundamental ideas that will go into the design of the University of Texas in San Antonio's robot entry into the 2007 CiberMouse competition. The requirement of the competition is to traverse unknown terrain with the goal of rescuing a stranded target and returning them to the starting location. The competition is handled through software simulation allowing us to focus all efforts on the algorithms necessary to achieve success.

1 Introduction

Several limitations arise when approaching a solution to this contest. The first is that there can only be two sensor requests made per cycle. This means that the robot needs to decide priorities for sensor requests based on specific events and states. Another limit is that the robot will need to maintain a map which can hold enough history so that it does not waste time revisiting areas it has already visited. This map can also be used to find the shortest path back to the starting position. Finally, the largest limit is positioning. Since the robot will not have access to a GPS sensor during the actual competition, there will have to be some sort of virtual positioning system based on the motor inputs. The ability of the robot to maintain an accurate map will depend greatly on the success of this positioning system.

2 Sensors

The sensors the robot can use include:

- One Bumper Sensor
- One Beacon Sensor
- One Compass Sensor
- One Ground Sensor
- Four Obstacle Sensors

They will be discussed in greater detail in the next sections.

2.1 Bumper Sensor

The bumper sensor notifies the robot of a collision which can occur with other robots or obstacles. The bumper sensor is available every cycle so the robot is able to check it in at all times. Upon recognizing that a collision has occurred, the robot will use a simple obstacle correction algorithm to put it back on track.

2.2 Beacon Sensor

The beacon sensor is used to identify the direction of the target relative to the robot. The beacon sensor has a latency of nine time units, so some bookkeeping will be required to maintain position and direction when fired so the robot can back track some to perform an exhaustive search for the signal. The sensor sits high enough on the robot so that it can see over low walls. It does not report the distance from the robot which the target lies.

Since the beacon sensor only returns a direction component and not a distance component it will not be able to pinpoint the target's location without at least two sensor readings.

2.3 Compass Sensor

The compass sensor reports the approxiate direction that the robot is facing. The compass is useful to correct the robot's bearing since its virtual positioning system will be based on motors that are impacted by noise. Since the compass sensor also has a latency of nine, the robot will have to maintain some additional bookkeeping for it too.

2.4 Ground Sensor

The ground sensor's only job is to check if the robot is over the target position. Once a positive reading has been received by the beacon sensor and the robot has entered the exhaustive search mode, this sensor will heavily requested in order to home in on the target's exact position.

2.5 Obstacle Sensors

Obstacle sensors report the distance from an obstacle they lie. They are subject to noise as well, so unfortunately they are only reliable for short distance readings. Finally, the obstacle sensors can be requested with zero latency. Their only constraint will be the

fact that the robot can only request two given sensors per round.

3 States

The attitude of our robot will be determined by which state it is in and its actions will be determined by which sensor events occur while in these states. These states will include *lost*, *obstacle tracing while lost*, *sighted*, *obstacle tracing while sighted*, *returning*, and *obstacle tracing while returning*. The following sections will describe the relative techniques that will be implemented by our robot within each of these states.

3.1 Lost State

The *lost* state is the initial robot state. The beacon sensor will be called very little in this state, while the primary sensors relied upon will be the bumper and obstacle sensors. Essentially, the first action in this state will be to shoot the beacon sensor. In the absence of the target's beacon being visible, the robot will assume that the target is behind a high wall and move forward to find the next obstacle to search behind. When it finds this next obstacle it shifts to the *obstacle tracing while lost* state.

3.2 Obstacle Tracing While Lost State

When the robot reaches an obstacle it will turn in a random direction and using the side obstacle sensor facing opposite that direction, make the necessary turns needed to follow the obstacles wall. Should it require a turn angle large enough to be considered to be a corner it should shoot the beacon sensor with the goal being to shoot the beacon sensor around every corner. Once the robot has made a corner it should stop following the wall and re-enter the lost state. The robot should switch states between *lost* and *obstacle tracing while lost* until the beacon sensor receives a visible reading. If the robot does find the beacon direction it should enter into the *sighted* state.

3.3 Sighted State

Making it to the *sighted* state is good news. The robot now knows the approximate direction of the target. Once the robot has received a beacon sensor reading it will change its direction to match the suggested target direction. Similar to the *lost* state, the beacon sensor is shot very little, focusing primarily on the results of the obstacle and bumper sensors. The robot maintains this state unless it finds an obstacle in which case it enters the *obstacle tracing while sighted* state.

3.4 Obstacle Tracing While Sighted State

In the *obstacle tracing while sighted* state, the robot maintains a similar wall following technique as in the *obstacle tracing while lost* state. Instead of deserting an obstacle once it has rounded a corner it follows the entire obstacle while shooting a beacon sensor around every corner. Essentially, it is in an exhaustive search mode during this stage. The robot only leaves the *obstacle tracing while sighted* state when it has re-entered an already visited block at which point it re-enters the *sighted state*.

If the robot has triangulated the position of the target, it can vary this algorithm to avoid wasteful searching. For example, it can identify when a target is not behind specific obstacles and can desert an obstacle when it resolves it to be non-useful.

3.5 Returning State

In the *returning* state the robot will figure out the direction from its current location to the start location and turn and drive toward it. The robot can enter this state from any of the prior states should the ground sensor get a successful reading. Should the robot encounter an obstacle it will enter the *tracing obstacle while returning* state.

3.6 Tracing Obstacle While Returning State

In the *tracing obstacle while returning* state the robot follows the obstacle until it turns a corner then returns to the *returning* state where it recalculates the direction to the starting location and turns to that direction.

4 Map making

In the absence of GPS the robot will be using the motor inputs to maintain its relative position on the map. Using this position system and a two dimensional grid based mapping system, it will maintain data to track visited areas as well as obstacle laden areas. This will aid in the obstacle tracing states so it does not retrace obstacles that have already been traced. It can also be helpful in the *returning* state's algorithm to allow for make a speedier return trip. For example, the robot can choose a route made up of locations it has already visited in order to avoid using a time consuming obstacle tracing algorithm.

The bookkeeping necessary for this operation will include an array that is several times larger than the actual arena size since the robot does not know its starting position. It will either hold a value that says not visited, visited, or obstacle.

5 The Return Home

To properly return to its starting position the robot will be utilizing a shortest path graph approach. The graph will be indirectly represented by a matrix overlaying the arena. Ideally each cell in the matrix will

represent a 1 um by 1 um area in the arena; however, this may take a large amount of memory as the number of cells will grow exponentially along with the size of the arena. Some cells may be skewed due to the approximated robot position.

In basic operation, each cell will be numbered in the order that it was visited by the robot. This will allow the robot to retrace all of its steps back to its starting position.

In advanced operation, simple paths with only one entrance and one exit can be merged into one edge with a weight corresponding to the number of edges that existed in that path. Then a shortest path algorithm like Dijkstra's algorithm can be applied to determine the best return route.

EvoRobert System Description

Ali M. Alanjawi
and Frank Liberato
Department of Computer Science
University of Pittsburgh
Pittsburgh, Pennsylvania 15260
Email: {alanjawi,frank}@cs.pitt.edu

***Abstract*—In this paper, we describe our approach to controlling a simulated mouse in the CiberMouse competition. Using Evolutionary algorithms to learn an effective control strategy based on inputs from various high-level sensor systems (constructed from the low-level hardware sensors provided by the simulator), we construct the control algorithm for EvoRobert.**

I. An Evolutionary Algorithm Approach

The CiberMouse competition requires that a robot navigate a maze in the presence of both sensor and motor noise. Because Evolutionary algorithms are known for their robustness in the presence of noise [MSG99], [HHG$^+$06], we decided to employ them to develop the high-level control strategy for EvoRobert.

A. SAMUEL

SAMUEL, which stands for Strategy Acquisition Method Using Evolutionary Learning, is a machine learning system that uses evolutionary algorithms to solve reinforcement learning problems. It explores decision policies in simulation and modifies those policies based on acts in the simulated environment. Policies are represented by if-condition-then-action rules. Consequently, it is appropriate for sequential decision problems such as controlling robot behaviors. SAMUEL supports parallel execution, making it a powerful tool for experimenting with robot behaviors and evolutionary algorithms. SAMUEL manual may be found in [Gre97] and [Dal05].

SAMUEL learns policies acting in a simulated environment by incorporating genetic algorithms and other rule-based learning methods (e.g., Lamarckian evolution[1]). The user decides the environment, provides the sensory information to its agents, and applies the actions of those agents in the environment as well as the fitness, after a number of trials.

The learning process begins by applying genetic operators to the initial population, which consists of the basic rules provided by the user. Then the population is evaluated before applying Lamarckian methods. The updated set of policies is evaluated again and the learning process is repeated until the user-defined ending criterion is met.

For our implementation, we modified the CiberMouse simulator to run faster in the learning process by adding JNI code into the simulator and compile it as a library for use with SAMUEL. Instead of sending sensor/action information over UDP, they are read directly from the simulator, hence cutting down the network and parsing overhead. Labs from previous years are also hardcoded in the modified simulator code, so SAMUEL can load a different lab at each evaluation.

The goal of each robot in the CiberMouse domain is get to the beacon and return back to its starting pooint as fast as possible. This task is far too complex to be learned at once. Hence, dividing the task into sub-tasks seems a natural choice. EvoRobert learns the different sub-tasks to accomplish CiberMouse goals. These sub-tasks are: explore, hunt, and return task; they are described below.

B. Exploration Task

At the starting point of the CiberMouse simulation, the robot might not sense the beacon, and thus, doesnt know exactly where to go. For this task, EvoRobert will try to learn how to explore the maze, until it can sense the beacon. EvoRobert can sense the surrounding walls, how much he explored, and whether he has been in the current location before or not. His actions are turning, and driving forward. The fitness function is designed to drive EvoRobert to successful behaviour in this task. If EvoRobert can sense the beacon at the end of his trial, he gets at least 50% of the payoff. If not, he gets payoff depending on how much he explored the maze. In early experiments, we noticed that EvoRobert would explore the maze well, however, he would return back to his starting point. So we added another entity to his payoff which adds the distance travelled as well. The final fitness, f, is calculated as

$$f = \begin{cases} 0 & \text{collided more than twice} \\ 0.5 + 0.5 \cdot (1 - \frac{t}{T_{MAX}}) & \text{finds beacon at time t} \\ 0.25 \cdot \xi + 0.25 \cdot \delta & \text{timeout} \end{cases}$$

where ξ is the percentage of the map explored, δ is the percentage of the straight line distance between the starting point and the beacon which EvoRobert reached, and T_{MAX} is the maximum amount of simulation time allowed.

In this task EvoRobert managed to get 100% of the payoff in the later generations in some of the experiments. It seems this task is easer than the other two! Figure 1 shows the performance of EvoRobert in this task.

C. Hunting Task

This is the main task that EvoRobert must learn. He must reach the beacon as quickly as possible. The obvious behaviour

[1]Lamarckian operators are different from Darwinian operators in the sense that the environment directly affects the organism's behavior rather than affecting only the selection course.

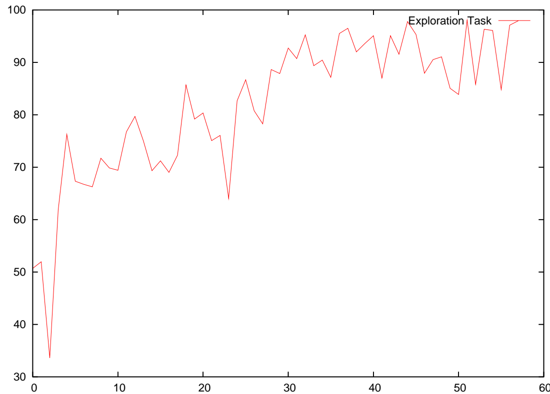


Fig. 1. Exploration task performance

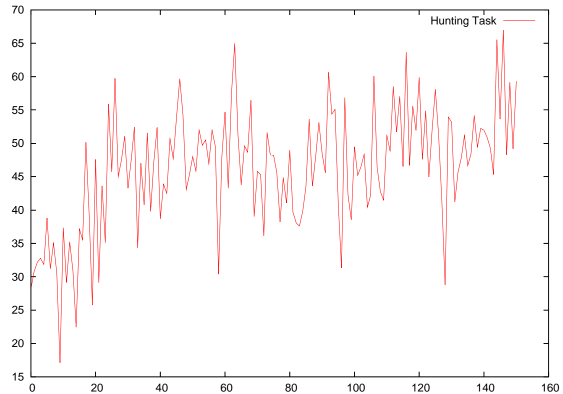


Fig. 2. Hunting task performance

is that EvoRobert adjusts his angle, and then drives straight towards the beacon. But that is not the case when walls are present. Therefore EvoRobert must learn how to go around a wall, e.g. follow a wall, until it finds the beacon. We decided to take another approach: let EvoRobert sense the direction of the shortest path to the beacon (possibly around walls), instead of sensing the actual beacon direction. Finding the shortest path to the beacon is discussed in Section III-C. This decision was made because it will cut down the learning time of EvoRobert drastically. The fitness is computed as follows

$$f = \begin{cases} 0 & \text{collided more than twice} \\ 0.8 + 0.2 \cdot (1 - \frac{t}{T_{MAX}}) & \text{reaches beacon at time t} \\ 0.5 \cdot \delta & \text{timeout} \end{cases}$$

where δ is as defined above. Figure 2 illustrates the performance of EvoRobert in this task.

D. Returning Task

This task is similar to the hunting task, except that now EvoRobert must sense the path to its starting point, instead of to the beacon. This is the only change that has been made from the hunting task. We have noticed that as time passes while EvoRobert is moving around the maze, inaccuracies in its calculations will accumulate, as described in section IV-B. To adjust for this, we simply flush all the information gathered during previous tasks.

II. Low-level Design Decisions

Given the rules learned via SAMUEL for the various tasks, we now must incorporate them into a realtime system that is capable of interacting with the original simulator. To do this, we decided to use a multi-threaded Java program. The *realtime thread* is responsible for interfacing with the CiberMouse simulator, while the *non-realtime thread* performs all the high-level sensor computations, and execution of SAMUEL rules. In this section, we describe the duties of the realtime thread.

A. Position and Angle Measurements

Using code from the CiberMouse simulator, we compute the changes in EvoRobert's position and orientation based on the left and right motor power. This computation is included in the realtime thread so that it is performed in step with the simulator. The estimates are made available to the non-realtime thread for use in many of its computations, as described below.

We do not account for drift due to motor noise.

At the beginning of the simulation, we sample the compass sensor to determine EvoRobert's true orientation. While this is obviously not necessary for navigation, it does provide efficiencies for the various mapping functions which will be performed by the non-realtime thread. Knowing EvoRobert's true orientation allows us to limit the maximum width and height of each map to be 28 by 14 robot diameters[2], thus reducing processing time. After this initial sampling, we use only our own estimate of EvoRobert's heading.

B. Sensor Layout

Unlike the standard CiberMouse sensor layout, EvoRobert uses all four IR sensors. Sensors 0 and 3 point roughly forward, at ± 25 degrees from north. Sensors 1 and 2 are situated at ± 90 degrees from north. We use two, rather than one, forward-facing sensors to reduce the 'blind spot' between the side sensors and the front. This enables us to move closer to walls, while reducing the chance that a wall corner will move out of the sensor coverage area during a turn.

C. Sensor Scheduling

The realtime thread is also responsible for scheduling all requests for sensors with the simulator, and compensating for the nine-cycle delay in the beacon sensor readings. At each simulation step, it records the internal estimate of EvoRobert's orientation a nine-entry circular buffer. It uses this information to adjust the beacon sensor reading so that it is relative to the current heading. We do not attempt to adjust for changes in EvoRobert's position.

[2]In practice, since we do not know the starting position, we double the length of both axes of all maps. For the duration of this paper, we will quote only the effective map size, and ignore this doubling.

TABLE I

SCHEDULING WEIGHTS OF EACH SENSOR FOR VARIOUS EVOROBERT TASKS. NOTE THAT EACH IR SENSOR COLUMN RECORDS THE WEIGHT GIVEN TO EACH SENSOR IN THE GROUP INDIVIDUALLY.

Task	Fwd IR	Side IR	Cmp	Bcn	Gnd
Compass Alignment	0.0	0.0	1.0	1.0	0.0
Exploration	0.5	0.4	0.0	0.2	0.0
Hunting	0.5	0.4	0.0	0.2	0.0
Hunting Near Bcn	0.4	0.1	0.0	0.0	1.0
Return To Start	0.5	0.5	0.0	0.0	0.0

Scheduling is performed using a fixed-priority scheme, inspired by [BL06]. The particular weights given to each sensor are determined by the current task which EvoRobert is trying to perform. The weights for each sensor during each task are shown in Table I. The non-realtime thread informs the realtime thread of the weights to be used, but it is the realtime thread which actually manages the scheduling. The non-realtime thread does not assume which sensors will be available on any given cycle.

In the case where an IR sensor detects an obstruction, sensor scheduling is modified slightly by the realtime thread. It marks that IR sensor as *urgent*, so that it receives higher scheduling priority while an object is close to EvoRobert. These additional urgent requests for the sensor are not counted against it when it becomes non-urgent; after the obstacle has moved out of range, the sensor resumes scheduling as if it has been scheduled normally for the entire time.

III. HIGH-LEVEL SENSOR INPUT

One job of the non-realtime thread is to use the low-level sensor information to construct *high-level software sensors*. Each of these sensors, described below, is intended to provide the Samuel system with sufficient information to perform the various tasks described above.

A. Exploration Map

During the Exploration task, the goal of the robot is to find a location from which the beacon is visible. To facilitate exploring the arena, we provide EvoRobert with a high-level sensor which indicates how much time the robot has spent at or near its current arena position. This map has a very low resolution of only 8 by 4 squares, since we do not care that EvoRobert explores every possible inch of the arena; we care only that it explores enough of it to receive a reading from the beacon sensor. Once we acquire such a reading, the Exploration task ends and the hunting task begins. The exploration map is not used after the Exploration task ends.

B. Beacon Localization

During the hunting task, EvoRobert has detected the beacon and attempts to move towards it. Because the beacon sensor is subject to noise, and also to errors due to changes in the robot's position during the nine-cycle sensor delay, we do not use any single point estimate of the beacon's direction for navigaion. Instead, we combine multiple beacon direction readings using a *probabilistic map* of the beacon's potential positions.

To do so, we use the fact that, given some beacon sensor reading s, we can approximate the probability (mass) function $p(h \pm \Delta|s)$ that the beacon lies along some heading $h \pm \Delta$ from the robot[3]. Accounting for the Gaussian noise included in s, we have

$$p(h \pm \Delta|s) \approx \int_{|h-s|-\Delta}^{|h-s|+\Delta} \varphi_{\mu,\sigma}(\alpha) d\alpha.$$

where $\varphi_{\mu,\sigma}$ is the Gaussian probability distribution function with the parameters given by the simulation. Ideally, we would choose Δ to match the angle difference between opposite corners of the map square we are considering, but in practice, we do not. We ignore the tails of the normal curve[4], where the noise is a multiple of 2π.

By assuming that the beacon is *somewhere* in the arena with *a priori* uniform probability, we normalize $p(h|s)$ to produce

$$p(\langle x,y\rangle|s) \approx p(ang(x,y)|s) \cdot \left(\sum_{\substack{-28 \le x' \le 28 \\ -14 \le y' \le 14}} p(ang(x',y')|s) \right)^{-1}$$

for any point $\langle x,y\rangle$ within the arena. $ang(x,y)$ denotes the angle from the mouse to (somewhere) inside the given map position, expressed relative to the mouse's heading. To combine $p(\langle x,y\rangle|s)$ with previous estimates, we simply make a few completely false independence assumptions, and let

$$p(\langle x,y\rangle) = \prod_{s \in \mathbf{S}} p(\langle x,y\rangle|s)$$

where $\mathbf{S}$ is the set of all known sensor readings[5] It should be stressed that, while we have made liberal use of approximations and just-plain-wrong assumptions, the result is generally sufficiently accurate to guide EvoRobert.

In order to use this probabilistic map, we find the maximum likelihood estimate of the beacon's position, which is simply the point $\langle x_{ML}, y_{ML}\rangle$ with the highest probability. We then use this position as a *virtual beacon sensor*.

When EvoRobert is within two robot widths of the estimated position of the beacon, it stops sampling the beacon sensor. This allows more sensor bandwidth for the ground sensor, while also reducing the effect of position changes during beacon sensor delay compensation. The sensor weights in use while EvoRobert is close to the beacon is given by the 'Hunting Near Bcn' entry in Table I. If EvoRobert moves more than two robot widths away from the beacon, then it switches back to normal 'Hunt' sensor weights.

[3] We ignore the effects of position changes here. Empirically, it does not seem to hurt performance much.

[4] In fact, we can approximate the whole expression with $cos(\alpha)$ for all forward-facing angles with reasonable results!

[5] In practice, we normalize this estimate to prevent the underflow caused by our independence assumptions. We also add a small constant to the $p(\langle x,y\rangle)$ estimate before normalization, to reflect the bias in our point estimates. Due to the drift in our internal position estimate because of simulator noise, plus the lack of compensation for position when using the delayed (real) beacon sensor readings, it is normally the case that our point estimates will be inconsistent. Also note that beacon readings are in no way independent, without knowledge of the beacon's position!

C. Path Finding and Obstacle Map

The virtual beacon sensor gives EvoRobert a goal at any particular step of the hunting task, once two beacon readings are found. To navigate towards this goal, we maintain a 56 by 28 *obstacle map*. When any IR sensor records a wall at close range (approximately half of the robot's diameter), we update our obstacle map to reflect a wall at that position. At this range, we found that the noise in the IR sensor does not affect obstacle avoidance much. This map is updated even during the Exploration task as IR sensor readings are collected.

Our choice of resolution for the obstacle map is based on the rules regarding arena layout. Since all openings in arena walls will be at least 1.5 robot diameters wide (i.e., three squares in our obstacle map), it is likely that any such opening will have at least one square in the obstacle map to represent it[6].

Given this map, we use a variant of Dijkstra's shortest path algorithm to compute, for the current goal (beacon) square, the shortest path from every other map square. These results are stored in a 56 by 28 *shortest path map*. To fill in this map, we start at the goal square (with a distance of zero), and work outwards to compute the distance to all other squares. Initially, we treat every non-goal square as having an infinite distance.

To update this estimate, we maintain a priority queue of map squares, ordered by shortest path distance to the goal. We remove the highest priority (closest) square from the queue, and consider each of the eight neighbors of this square on our shortest path map. We compute the distance to the goal from this neighbor going through the dequeued square, by adding the straight line distance from the dequeued square to the neighbor to the shortest path distance of the dequeued square. If this is smaller than the shortest path distance recorded for the neighbor, then we update the shortest path map, and queue the neighbor for processing later. If it is not, then we discard this longer path, and do nothing additional for this neighbor. For neighbor squares which are marked as walls in the obstacle map, we never update the shortest path distance. We process map squares from the priority queue until it is empty[7].

Given the shortest path map, we provide a compass-like sensor to the Samuel rules which always points along the steepest downhill gradient of the eight neighbors from EvoRobert's current position. As the IR sensors update the obstacle map, the shortest path map is recomputed to account for them.

EvoRobert's starting position is used during the return task as the goal position.

[6]Note that at a range of one half robot's diameter, the width of the IR sensor beam s about 0.577 robot diameters. The walls bordering a 1.5 robot diameter opening might consume an extra map square due to rounding, plus an additional extra square because of the width of the IR sensor beam. We blissfully ignore the effect of non-axis-aligned walls, much like we neglect the interdependence of beacon readings in the previous section.

[7]For efficiency, if we update a neighbor which is the current position of EvoRobert, we also stop the whole shortest path computation. While this does not stricly guarantee that we will find the shortest path from EvoRobert to the goal, in practice it saves a large amount of computation time while not seriously affecting the result for arena layouts that we consider to be likely. We also omit the simple proof that this algorithm stops even without this optimization.

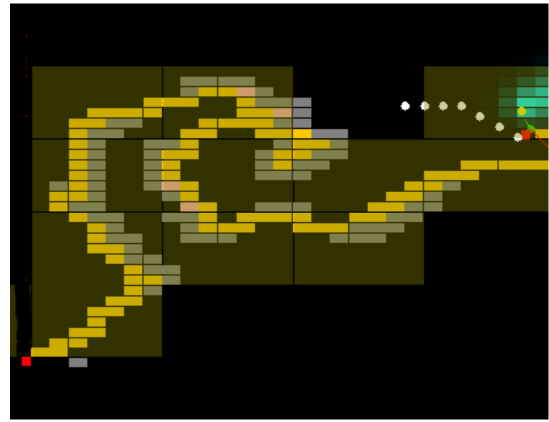


Fig. 3. Java-based visualization tool

IV. Testing and Implementation

A. Visualization Tools

With this myriad of sensors, we found it useful to have a visualization tool for them. A screenshot of this Java-based tool is shown in Figure 3.

B. Drift

Because of our reliance on our internal estimate of EvoRobert's position to maintain the various maps, drift due to motor noise can be a potentially large problem. While usually the overall effect is not large, on some runs it can be huge. This is very noticable if EvoRobert's estimate of its angle is incorrect. If drift is disabled in the simulator, EvoRobert maintains position and heading estimates accurately.

V. Future Work

We believe that drift is the single largest source of variability in EvoRobert's performance. To correct for it was beyond the scope of this project. Finding an effective way to compensate for it could make EvoRobert significantly more reliable.

We would like to perform a sensitivity analysis on several constants we used in this project. It is likely that EvoRobert can be 'better tuned' than it is now. Finally, we have ignored the effect of other robots in the simulation. A co-evolutionary experiment between more than one agent would be interesting.

References

[BL06] Bjorn Brandenburg and Hennadiy Leontyev. Ramses the rat: Cibermouse agent. *RTSS Workshop CiberMouse*, 2006.

[Dal05] Robert Daley. *JAGA-SAMUEL A Java Implemenatation of John Grefenstette's SAMUEL Learning System User Manual*. Navy Center for Applied Research in Artificial Intelligence, 2005.

[Gre97] John Grefenstette. *The User's Guide to SAMUEL – 97: An Evolutionary Learning System*. Navy Center for Applied Research in Artificial Intelligence, 1997.

[HHG+06] Guo-Sheng Hao, Yong-Qing Huang, Dun-Wei Gong, Guang-Song Guo, and Yong Zhang. Fitness noise in interactive evolutionary computation and the convergence robustness. In *ISDA '06: Proceedings of the Sixth International Conference on Intelligent Systems Design and Applications (ISDA'06)*, pages 429–434, Washington, DC, USA, 2006. IEEE Computer Society.

[MSG99] D. Moriarty, A. Schultz, and J. Grefenstette. Evolutionary algorithms for reinforcement learning. *Journal of AI Research*, 11, 1999.

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 sill 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.

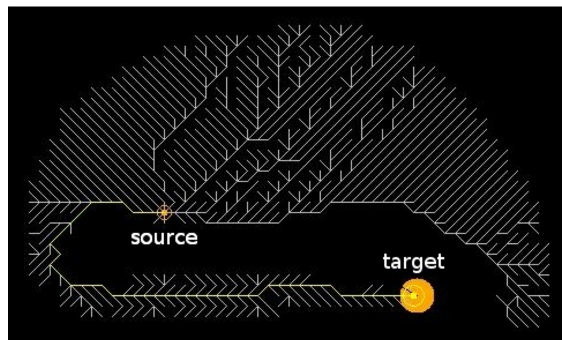


Fig. 1. Points visited around a wall, with lines to parent nodes in A*. The path is searched from the source to the target.

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:

$$\begin{aligned}&State\ \{\\&\quad bumper,\\&\quad left_motor,\ right_motor,\\&\quad beacon_distance,\ beacon_angle,\\&\quad obstacle_sensors[n]\\&\}\end{aligned}$$

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 then 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

FAUBOT: Purposeful Navigation of a Robot in a Simulated Environment

Daniel Danner, Caroline Kaufhold, Peter Kranz, Rainer Müller, Sven Pfaller,
Christian Rieß, Elli Angelopoulou

Department of Computer Science, University of Erlangen-Nuremberg, Germany.
{sidadann, sicakauf, sipekran, sirrmuel, sisvpfal}@stud.informatik.uni-erlangen.de,
{riess, elli}@i5.informatik.uni-erlangen.de

***Abstract*—In the CiberMouse Contest a robot has to find in a simulated environment its way to a beacon and back to its initial position using data from different sensors. This paper presents the design of the FAUBOT and the challenges the team met when creating the robot. Particular algorithms and datastructures that are used include Brook's subsumption architecture, the A* algorithm and a probability map of the world.**

I. INTRODUCTION

The CiberMouse simulation environment is a platform to experiment with virtual robots and is for example used in the annual Ciber-Rato Contest [1]. Since the setup is bound to short simulation cycles, special attention has to be paid to real-time performance. The CiberMouse organization provides the technical framework to start right away with the development of algorithms. In general, the task is to direct a mouse through a maze to find a beacon. The mouse is modeled by a virtual robot that has sensors and actors. For orientation, the robot has four free-to-place obstacle sensors (the "eyes") and a beacon sensor (the "nose"). For navigation, it has two individual controllable wheels. Also, for test and evaluation purposes, there is a GPS tracker installed to externally monitor the actual position. The task is not only to find the beacon but also to find the way back to the initial position (in the following referred to as "home") and to be faster than the two other competing mice. Therefore, three tasks have to be solved:

1) The sensor data has to be handled properly and some movement strategy has to be developed to build a reliable map.
2) Once a (partial) map is available, a sophisticated pathfinding algorithm needs to be employed to find the way to the beacon and back.
3) The robot must not collide with walls or other competing robots.

This paper is structured as follows: First, we give a overview of the used algorithms. In section III the framework and our robot system architecture is described in detail. Several other aspects that came up during the development of the robot are addressed in section IV, as well as the overall system performance. We conclude with a short summary and an outlook.

II. SYSTEM OVERVIEW

The FAUBOT needs to explore the environment randomly as long as there is no available information about the beacon's position. Until the beacon is found, the robot is continuously exploring the world. During these operations, a probabilistic map is built up based on the perceived sensor data. It contains not only supposed obstacles but also visited areas which will be assumed "safe for driving" later on. As soon as the beacon is detected, we rely on naive driving to the measured direction while obstacle avoidance is handled using a non-deterministic algorithm.

When the beacon has been reached, the FAUBOT uses a modified A* algorithm on the probabilistic map in order to compute the optimal known path home through the maze. As long as the robot moves along areas that are marked *visited*, we know that appearing obstacles are other robots. The avoidance

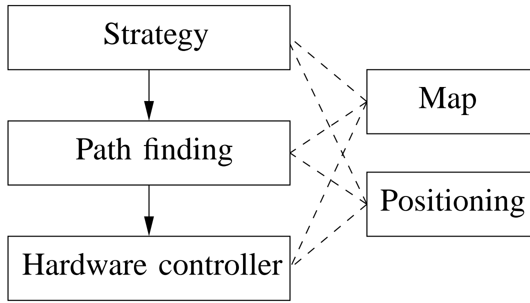


Fig. 1. System architecture: The components on the left form a hierarchical decision module. Map and positioning component provide the fundamental data for those decisions.

routine is a combination of the wall avoidance in the exploring phase and the A* algorithm.

III. System Architecture

The system architecture is split into three major parts, "strategy", "path finding" and "hardware controller", as shown in fig. 1. They build a vertical hierarchy of different abstraction levels over which sensor data and commands are passed upwards and downwards.

Sensor data is updated in discrete time units called "cycles". The robot can read two sensors in every cycle, which demands a trade-off between the demand of different sensor data. After reading the sensors of the current cycle the components are updated with the new information, and start to perform the higher level computations. The first component to be considered is the strategy which issues commands at the highest level of abstraction. On the second level the path finding component executes the strategy's commands by autonomously performing obstacle avoidance. The lowest level provides a basic sensor scheduling and the evaluation of the sensor data. Additionally, higher level components can always override decisions of the lower level components. This organization of tasks is similar to Brooks' subsumption architecture [2].

Besides those three layers there are a "map" and a "positioning" component, which provide a representation of the world and the robot in it. Those are both independent of the operating hierarchy and compute the information based on the data we get from the sensors.

In the following we give a description of the different layers, their domains and interactions.

A. Strategy

The strategy component internally implements a finite state machine. It basically consists of three states that describe our main goals. At start-up the FAUbot is not aware of the beacon position, so it has to *explore*, which is the first state. The strategy component selects waypoints where the bot should drive to in order to find the beacon. When the robot spots the target it switches to the next state, *drive to beacon*. The strategy decides between two alternatives:

- driving on a straight line towards the beacon. Note that once the strategy decided to head directly towards the beacon, the responsibility for driving along the line and to the assumed target position relies on the path finding component, including the handling of walls and obstacles.
- gathering more precise data about the beacon position. By collecting multiple measurements of the beacon sensor that are represented as lines between different robot positions and the beacon, we can compute where those lines intersect. The strategy then assumes that the beacon is located in the area where the intersections accumulate.

In order to get more quickly to the target, mostly the first alternative is chosen.

Once the beacon is reached, the strategy switches to the *return home* state. Based on the data collected so far, the strategy instructs the path finder component to drive to the start position, where internally an A* algorithm is used.

The whole state machine is modularly designed so states can easily be added or removed. For example, additional states can be implemented like a higher level strategy for the avoidance of other robots.

B. Path finding

The path finding component is comprised of the following three methods:

- **drive to point:** The path finding algorithm heads directly or in small slopes for a given (x, y)-coordinate. When an obstacle occurs it uses a modified *Wall Follower* algorithm by randomly avoiding it to the right or to the left. This method is used during the exploration phase, and when the assumed beacon position

could be sufficiently narrowed down by examination of multiple signals from the beacon.
- **drive to beacon:** As long as there were too few beacon signals received for narrowing down the beacons position, only an approximate direction towards the beacon is known. The algorithm leads then directly along the bearing. The obstacle avoidance is similarly done as in the first case. One addition is to look around for further beacon signals once the robot left the direct trail towards the beacon.
- **drive home:** The last method is returning to our initial starting position. As there is already at least one known complete path from the home to the beacon from the exploration phase, we use a modified A* algorithm [3] on the collected data to compute a way back.
  The algorithm does not use the full resolution of the internal map, but operates on median filtered cells that are larger than our internal representation. This ensures that the computation of the A* algorithm finishes within a few cycles, and brings some additional security distance to the walls, since the lower resolution slightly increases known obstacles.

The strategy component can at any time override the behaviour of the path finding component and issue modified or different commands.

C. Hardware controller

The hardware layer provides a reliable abstract sensor and actuator API for the higher level methods. This comprises

- **engine actuators:** It is only possible to set a certain amount of power to the engines, thus it is necessary to estimate the actual speed. The FAUBOT does this by applying the acceleration formula given in the rule set to the difference in the power supply between the last time step and the current one. In the development phase this formula was checked by comparing the current position plus the speed estimates against the GPS coordinates, which yielded sufficient results.
- **beacon sensor:** The beacon sensor has a significant latency. As soon as the beacon is detected this event is applied to a state in the past and stored. The last ten beacon sightings are used to determine the area where the beacon is assumed to be located.
- **obstacle sensor:** The FAUBOT uses the standard distribution of the obstacle sensors around the robot, where the rear sensor is not used. When an obstacle is detected, an entry into the map is made based on our assumed position. In order to avoid collisions the front sensor is higher weighted than the side sensors.
- **compass sensor:** Similarly to the beacon the compass has a high latency. Thus the positioning using the compass can be used on data in the past, since the FAUBOT incorporates the compass data online, and does not stop and wait for new compass data to arrive. This causes a reevaluation of all positioning results in the past nine cycles, and therefore also a reeveluation of almost all other recent results.
- **ground sensor:** If the FAUBOT is driving directly towards the beacon, the ground sensor is activated and regularly queried.

Most work of the sensor data evaluation is about integrating the obstacle and compass sensor together with the engine actuators into a common, accurate map, thus the hardware controller is closely connected to the map and the positioning components.

D. Positioning component

A reliable estimation of the robot's position in the world is essential for all decisions. This task is implemented in the positioning component. Its computation is based on assumptions of the motor behaviour taken from the rule set. In order to minimize our calculation error the compass is used to adjust our angle and therefore our position. The high latency of the compass requires us to apply these corrections to data that were collected several cycles earlier.

E. Map

The internal representation of the world is built up in the map component. At first it gets the raw data from the sensors that were queried in the current cycle. These are preprocessed and added to the map of the world included into this component. In order to provide as reliable as possible information for the modules that control the actions of the FAUBOT, noise has to be taken under consideration.

This is done by normalizing the received sensor data using a Gaussian distribution. The area that is covered by a specific sensor is updated with certain probabilities for containing an obstacle or not. Additionally the map marks visited sites in the map as *free*. Whenever an obstacle shows up in free areas, it can not be a wall, but one of the other robots.

IV. Experiences

A. Visual debugging

One of the first things we built up during development was a graphical interface for allowing easy debugging of the various low-level components. It turned out to be very useful in comparing the assumed position with the GPS coordinates, visually verifying the probabilistic map against the actual maze, and for displaying the generated log messages in an integrated environment. We also built in a mode for manually controlling the FAUBOT through the map in order to evaluate the map's robustness with respect to drawing walls and the effect of passing competing robots. In fig. 2 an example of our graphical representation is given. The image on the bottom shows the given simulation environment. The image on the top shows a visualization of the generated probability map, where darker grey levels correspond to higher obstacle probabilities. In this specific case the robot obviously did not manage to get past the second barriers.

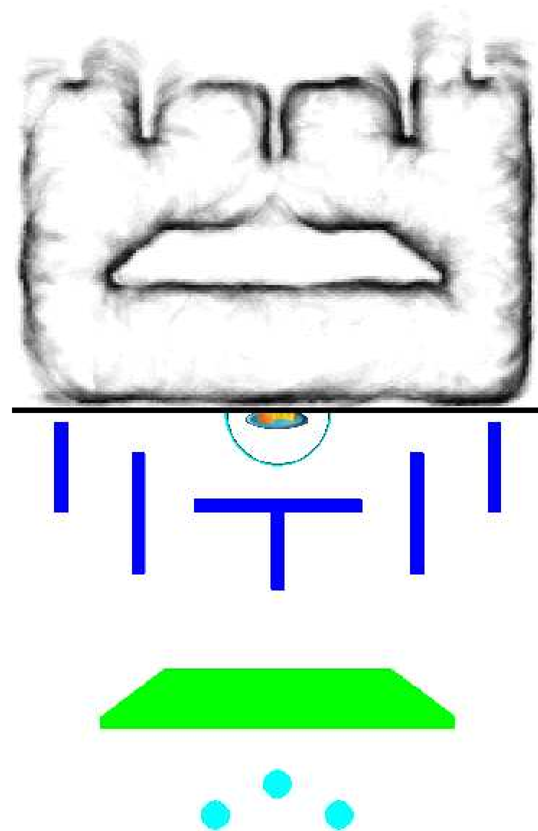


Fig. 2. Comparison between a given maze and the robot's representation of the world being partly explored. In this case the beacon was not yet considered.

B. Development problems

We tried to use a similar approach for the computation of the beacon's position as we did for the obstacle detection. This required entering for every direction of the beacon's signal a 4-degree circle sector into the map within the maze's boundaries. Unfortunately we did not succeed to perform this computationally efficiently, thus we dropped that approach.

Our geometric solution of representing the detection of a beacon as a line between the robot and the beacon position seems to produce also acceptable results, though we believe the method described above leaves some room for improvements.

V. Conclusion

Looking back, the modular approach has been successful as the behaviour can be easily changed in individual components. Additionally the graphical environment proved to benefit the development process, especially since the robot simulation must be written in comparably short time.

Currently our robot is fulfilling the basic requirements for the contest like finding the beacon and returning back home. We are sure several optimizations of our strategy or algorithms are possible and we will try to integrate them until the event. We are eager to test the performance of the FAUBOT in the CiberMouse competition at RTSS 2007.

References

[1] N. Lau, A. Pereira, A. Melo, and A. Neves e João Figueiredo, "Ciber-Rato: Um Ambiente de Simulação de Robots Móveis e Autónomos," in *Revista do DETUA*, vol. 3, no. 7, 2002, pp. 647–650.

[2] R. A. Brooks, "A Robust Layered Control System for a Mobile Robot," in *Robotics and Automation, IEEE Journal of*, vol. 2, no. 1, March 1986, pp. 14–23.

[3] P. Hart, N. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," in *Systems Science and Cybernetics, IEEE Transactions on*, vol. 4, no. 2, July 1968, pp. 100–107.

RoboCoog: The CiberMouse Agent

Ashish Kapadia, Arun Chhetri, Ronak Shah, and Dr. Albert M. K. Cheng (Advisor)
ashishbk@gmail.com, achhetri@uh.edu, rshah10@uh.edu , cheng@cs.uh.edu
University of Houston – Department of Computer Science

Abstract: *RoboCoog is the name of the robot from University of Houston that will participate in the RTSS 2007 CiberMouse competition. The competition will be held in discrete time driven simulation environment where the responsibility of each robot is to find a beacon in an unknown environment and come back to the starting position of the robot before the other robots. At the same time, the robot should try to avoid collisions with other robots and obstacles in order to avoid penalties. In this paper we describe our approach to solve various challenges posed by this competition.*

1. Introduction

The main challenge in the CiberMouse competition is to use various sensors available to the robot to learn the unknown environment that it needs to explore in order to reach the goal. The noise in the values provided by the sensors makes the problem particularly difficult and closer to real-world problem faced by the actual robot. The robot is given few actuators via which it interacts with the environment.

We have identified various sub problems that we will face in order to build the robot. We are listing our approach to solve these problems in the subsequent sections. Section 2 describes the approach we are taking to build the map in order to remember the unknown environment that we explore. Section 3 describes various states we've identified the robot will be in as it attempts to fulfill its goal. Section 4 describes utilizing the scarce sensor resources in order to efficiently manage them. Section 5 will describe our strategy on avoiding the obstacles. Section 6 will describe our approach on locating beacon. Section 7 will describe return strategy employed by the robot to come back to starting position.

2. Map Building

In order to remember various places that robot has visited it is essential that robot builds a map so that it can avoid already visited area in the map. In addition to this, map is essential for the competition since the robot needs to come back to the position from which it started.

Since the size of the arena is known to be 28 unit length wide and 14 unit length long, we can represent the map in a two dimensional array. Various experiments on the simulation environment have provided us data on how many unit length our robot will move when we apply specific power to the motors. Based on this, we've determined to use 0.1 unit length for each cell in the array. In addition to this, since we do not know the actual position of the robot in the map during the start of the competition, we've decided to double the original width and height of the map to be 56 unit length and 28 unit length respectively. Therefore, our map

will be two dimensional array with 560 width and 280 length. The figure 2.1 shows how this configuration takes care of possible extreme cases in which the robot may be placed in the map.

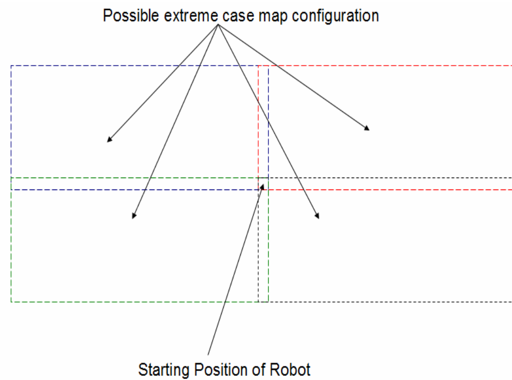


Figure 2.1. Map configurations.

One simple way to represent the map is by marking the map with various states such as: UNKNOWN, VISITED, OBSTACLE, BEACON etc. Even though this approach may work, it does not take noise into account. In addition to this, since there are other robots that are moving in the maze we must also consider a possibility that we may incorrectly mark them as obstacle.
A better approach to this problem is to use probability value (belief value) to mark cells as obstacle. On multiple readings by the robot, the robot can increase or decrease these cell values based on whether the cell has obstacle. This probabilistic approach of building the map would help when another robot is incorrectly marked as obstacle. It will also take multiple measurements of the cells into account.

The probabilistic map will help in dealing with noise in some aspect but it is not going to totally eliminate the major problem related to cumulative noise caused by noise values in the motor. Since the noise is going to be in Gaussian distribution, one can use Kalman filter algorithm to reduce the localization error caused by noisy error. The Kalman filter works particularly well in dynamic system with noisy measurements.

3. Robot States

The following figure shows the states of the robot.

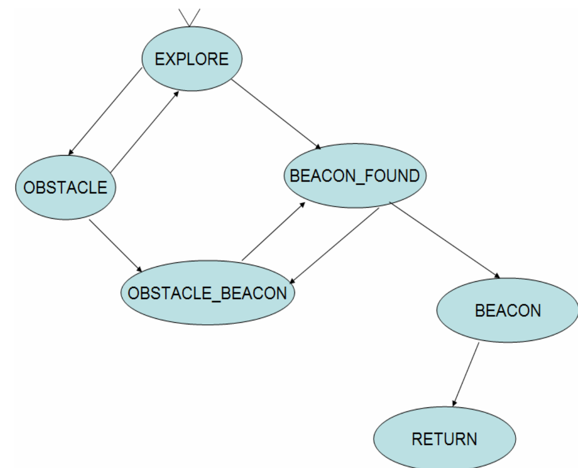


Figure 2.2. States of the robot.

EXPLORE State: This is the starting state of the robot when the robot has not sensed beacon or obstacle. The robot remains in this state until it senses either the beacon or an obstacle. In this state the robot tries to explore the environment and based on information stored in Map, it avoids visiting the same area twice.

OBSTACLE State: The robot comes in this state when it is in explore state and sense an obstacle. In this state the robot avoids the obstacle and then goes back to EXPLORE state to explore more

environment. When the robot senses the beacon in this state it will go to OBSTACLE_BEACON state.

OBSTACLE_BEACON: In this state the robot has the knowledge of the location of beacon but there is some obstacle in between the two. The main aim of robot in this state is to avoid the obstacle and move in such a way that it goes close to the beacon.

BEACON_FOUND: The robot comes in this state from either EXPLORE state or OBSTACLE_BEACON state. In this state the robot only senses the beacon and not obstacle.
BEACON state: This state indicates that the robot has reached the destination and it is on the beacon. This state can only be reached from BEACON_FOUND state.

RETURN state: The RETURN state indicates that the robot is returning back to staring position. Section 7 describes our strategy on returning back to starting position.

4. Sensor Scheduling

We have implemented different scheduling strategies depending upon the state the robot is in. The priorities of different sensors change as we move on from one state to another. For example in EXPLORE state the robot will initially use the beacon sensor in order to locate the beacon so that robot can immediately go into BEACON_FOUND state. We will schedule the beacon sensor after every 9 cycles with highest priority. In the mean time the robot will use center obstacle sensor and then alternatively left and right obstacle sensors to check whether there is any obstacle in the path. If there is an obstacle then the robot will go into OBSTACLE state.

5. Obstacle Avoidance

The key decision to make when avoiding the obstacle is to decide which direction the robot should move. This decision is made based on taking several factors into consideration such as querying the map to determine the best next move or with the help of left and right center we may be able to detect a non-obstacle on either side.

6. Locating Beacon

Correctly estimating the position of the beacon is an important aspect of the competition. The location of the beacon can be estimated based on previous readings from various location of the robot. We are still exploring the best way to precisely estimate the location.

7. Return Strategy

As we were moving from start position to target place we were building map at each instance by considering all obstacles that we encountered in our path. By using this build map we know which is start position from where robot started the competition and using this details available in the map we can find the path by which robot can come back to start position in such manner that it does not need to visit entire area which robot visited while come to beacon.

9. Conclusion

The CiberMouse competition is an interesting challenge that exposes various problems that are faced by building an actual robot in the simulation environment. The sensor noise particularly makes the task difficult. We look forward on applying the strategies we have described in here in the actual competition.

References

1. Bjorn B. Brandenburg Hennadiy Leontyev. Ramses the Rats: Ciber Mouse Agent http://www.ieeta.pt/~lau/web_ciberRTSS/docs/RamsesTheRat.pdf
2. Leonid Ryzhyk, David Snowdon, and Stefan Petters. Numbat: an entry to CiberMouse@RTSS2006. http://www.ieeta.pt/~lau/web_ciberRTSS/docs/Numbat.pdf
3. Ciber-Mouse: Rules & Technical Specification. http://www.ieeta.pt/~lau/web_ciberRTSS07/docs/ciberRTSS07_rules.pdf

SWIFT – A Scenario based dynamic sensor scheduling robot

Jaya Shanmukha Srikanth Palli, Krishna Konda
Department of Computer Science, Department of Electrical & Computer Engineering
Texas A&M University
College Station, TX 77840
pjssrikanth@gmail.com, kkchaitu@yahoo.com

Abstract - **In this paper, we discuss about the issues faced and the approaches we used to develop the virtual robot -- SWIFT, a scenario based dynamic sensor scheduling robot. Our robot is based on the following strategies: Wall Following Approach, Virtual coordination system, Beacon following, Coordinate Triangulation to predict the beacon location and following the Slope for backtracking.**

I. INTRODUCTION

CiberMouse 2007 is a virtual robotics competition held as a part of the 28th IEEE Real-Time Systems Symposium. The virtual environment is provided by a simulation system which creates a virtual maze with a target area along with a beacon to guide the robot agent to the target area impeded by obstacles along the way. The goal of the robotic agent is to navigate around the maze and reach the target area and return to the return as close as possible to the starting position in the maze. It must achieve this by avoiding collisions with obstacles along the way and other robotic agents in as small duration as possible.

The simulation system consists of the simulator engine, the visualizer and the robotic agent. The simulator manages the state of the simulated world by moving the robotic agents according to their motor commands and providing sensory inputs to them according to their position in the maze. The simulation system is discrete and time-driven. During each time step the simulator sends sensor measurements to agents, receives actuations orders, applies them, and updates scores. The visualizer is responsible for graphical display of the robots in the virtual maze, including their states and scores.

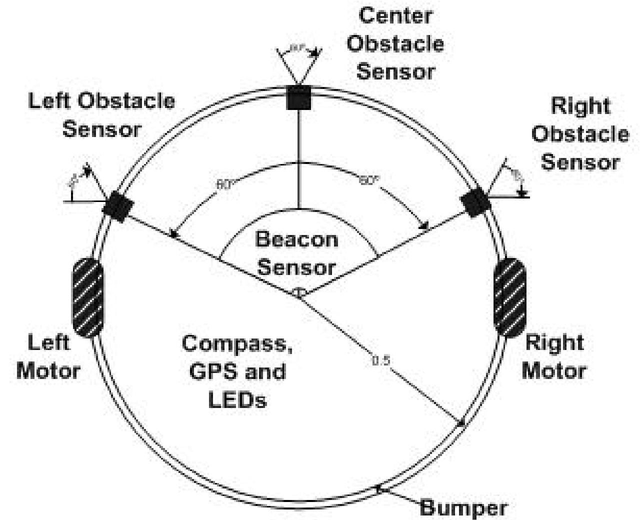


Figure 1. Diagrammatic Representation of the Virtual Robot

The robot is equipped with obstacle sensors, beacon sensor, compass, collision sensor and a ground sensor. The sensor availability is limited to two sensors per cycle. The robot has two motors which drive its left wheel and right wheel respectively. Both translational and rotational movements are possible, depending on the power applied to each wheel. There are three LEDs, named Beacon or Visiting, Return and End, which are used to signal the three goals. The contest deliverable is the robotic agent which would be evaluated in various mazes to see how well it performs.

The rest of the paper is organized as follows. Section II describes the various challenges that need to be addressed while developing the navigation algorithm. Section III describes the different stages of the navigation strategy and explain how the robotic agent would perform under each stage.

II. CHALLENGES TO BE ADDRESSED

This section describes the various challenges involved in the development of the navigation algorithm for the autonomous robot (mouse) in an unknown maze.

A. *Sensor scheduling*

The robot can request only two sensors value per simulation cycle, out of the seven sensors present in the robot – this puts a greater importance on how sensors are scheduled in this constrained space. While designing our robot, we focused a lot on creating a scenario based schedule.

Depending on the circumstances of the simulation cycle, different sensors should be scheduled. For example, when very close to an obstacle, in order to avoid a collision, we should schedule in the IR (infrared) sensors in that particular direction. In case there is no obstacle detected, then we can schedule the beacon sensor and determine in which direction the destination lays. These are some of the issues to address when scheduling the sensors.

B. *Collision avoidance*

Reliable obstacle avoidance is a vital feature, as penalties for colliding with static / moving objects accumulate during every round, therefore punishing the competing robot if it touches objects in the maze repeatedly. In case of a collision, the penalty is 5 points. Due to the high penalty, collisions with either the obstacles or other robots moving around the maze must be avoided or minimized as much as possible. This depends greatly on the obstacle sensor thresholds and beacon sensor values. Since the beacon sensor value is delayed by 9 cycles, this involves keeping track of the last 9 cycles and adjusting the movement in the direction of the beacon since the reading has been requested.

C. *Reaching destination*

In order to the reach the destination, wall following approach along with the beacon direction and the triangulation approach are used. This technique enables the robot to get around a wall and keep going towards the destination (the target indicated by the beacon). It follows the wall, until it comes around the wall and is able to navigate towards the target.

We have a few check-points in our wall following routine to ensure that the robot does not follow the same wall indefinitely. In order to do this, the robot periodically spins 360 degrees to look for the beacon. If the beacon was seen during this spin, it backtracks to the direction in which it saw the beacon. The robot then moves straight ahead in the direction of the beacon. If the robot has gone a full circle around the wall and still had not seen the beacon, we would break out of wall following and move in the unvisited direction. As the robot navigates through the maze, virtual coordinates are generated in the maze, which assists the robot eliminate navigations in the visited directions.

D. *Return Strategy*

Returning as close as possible to the original position in the maze is a very important part of the competition. Since there is no beacon or any similar means to help us reach the destination, some form of virtual coordinates system should be used in order to reach the original position in the maze. Our robot can get the destination's direction, by computing the tangent of the line connecting the current position and destination. Even while returning, the robot takes care to avoid obstacles and incorporates wall following to get around a wall. The collision avoidance and wall following approaches previously discussed are reused in this algorithm.

E. *Noises*

Each sensor has additive noise associated with it. This noise should be taken into account if nearly accurate results are to be obtained.

The following sensors have noises associated with them

1. Obstacle sensor (infrared sensors): Each obstacle sensor reading has a small amount of noise associated with it. Identifying the magnitude of the noise component for each sensor reading is important.
2. Motor noise: When power is applied to the wheels, there is a small noise associated with the power that is actually applied to the wheels. This will impact the localization information maintained and that information if not corrected for this noise, there is a chance the robot will not reach the originating position in the maze.
3. Compass noise: When using compass to navigate in the direction of the beacon or in the originating direction, compass noise if not corrected can lead the robot astray.
4. Beacon noise: Since the beacon helps the robot achieve the first goal, the noise must be smoothened out or else the robot may never reach the destination.

III. NAVIGATION ALGORITHM IMPLEMENTED

The three goals are

1. Identify the destination and reach it
2. Identification of the completion of goal 1
3. Return to the originating position.

The first goal is achieved by use of obstacle avoidance, triangulation and wall following combined with reading the beacon sensor and modifying the direction of the robot to point in the direction of the beacon. This does not always lead to the desired results since there can be obstacles that can cause the robot to move in infinite cycles preventing the robot from reaching the destination. A cycle avoidance scheme is used to detect and break cycles so that the robot does not end up in infinite loops.

The second goal is achieved by reading the ground sensor and setting the appropriate LEDs after having reached the destination and the return journey should be marked by the robot.

The third goal is reaching the originating point in the maze. This is achieved by use of virtual coordinate system that will help position the robot in the maze and also identify the direction of movement of the robot in order to reach as close as possible to the originating position.

Collision avoidance and wall following are the techniques used in all the three goals. Obstacles can be identified by setting thresholds for the obstacle sensor readings and being able to identify when there is an obstacle directly in front of the robot, when the obstacle is far away but in the direction of the beacon and wall following need to undertaken until the wall ends and we

can move in the direction of the beacon. Wall following is implemented by using both right strategy and left strategy as shown in figure 2.

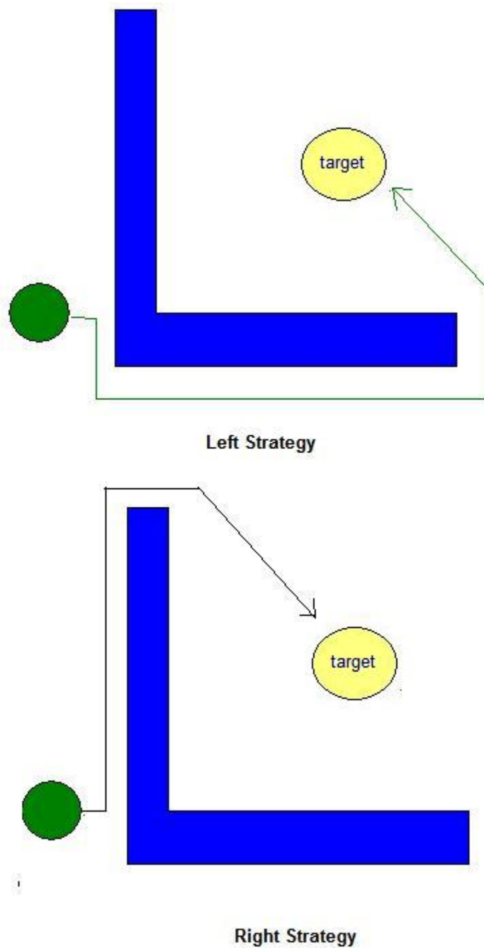


Figure 2. Left and Right wall following schemes

The localization information that is to be used in the return journey is created during the first and second part of the goals and updated during the final goal. The localization information is calculated from the following equations

$lOutPow_t = (lOutPow_{t-1} + lInPow_t) / 2$

$rOutPow_t = (rOutPow_{t-1} + rInPow_t) / 2$

$lNoisyOutPow_t = lOutPow_t * lNoise_t$

$rNoisyOutPow_t = rOutPow_t * rNoise_t$

where

- $lInPow_t$ and $rInPow_t$ are the power values supplied to the simulator at time instant *t*
- $lOutPow_{t-1}$ and $rOutPow_{t-1}$ are the power values produced by the motors at time instant *t-1* i.e. previous simulation cycle
- $lOutPow_t$ and $rOutPow_t$ are the power values produced by the motors at time instant *t* i.e. current simulation cycle
- $lNoise_t$ and $rNoise_t$ are randomly calculated motor noise
- $lNoisyOutPow_t$ and $rNoisyOutPow_t$ are the power values to be applied to the wheels at time instant *t*

These are the equations used by the simulator in calculating the power to be applied to the wheels based on the power supplied to the motors. From these equations, linear and rotational movements can be calculated by using

$lin_t = (lNoisyOutPow_t + rNoisyOutPow_t) / 2$

$rot_t = (rNoisyOutPow_t - lNoisyOutPow_t) / diam$

where

- lin_t and rot_t are the linear and rotational components of the movement at time instant *t*
- diam is the diameter of the robot

Using the above equations, localization information can be calculated as follows

$X = \Sigma (lin_t * \cos(rot_t))$

$Y = \Sigma (lin_t * \sin(rot_t))$

The above equations will help keep track of the current location which will be retained and used to detect cycles or multiple visits to the same location or nearby location and even the maze map can be created indicating where the obstacles lie in the map.

Once the coordinates have been calculated triangulation can now be used after two the beacon sensor values are received. Coordinates of both the locations where the beacon was requested is known - using these coordinates and the angle made by lines joining the beacon with these two points, the beacon coordinates, distance and angle from the present location can be calculated using triangulation. Using this information and the motion information from the previous 9 cycles (when the beacon was requested), direction of travel can be modified so that the robot moves in the direction of the beacon.

Triangulation, in trigonometry and geometry, is the process of finding coordinates and distance to a point by calculating the length of one side of a triangle, given measurements of angles and sides of the triangle formed by that point and two other known reference points, using the law of sines. For example, in the triangle shown in the figure 3, angles α and β are known along with the distance AB.

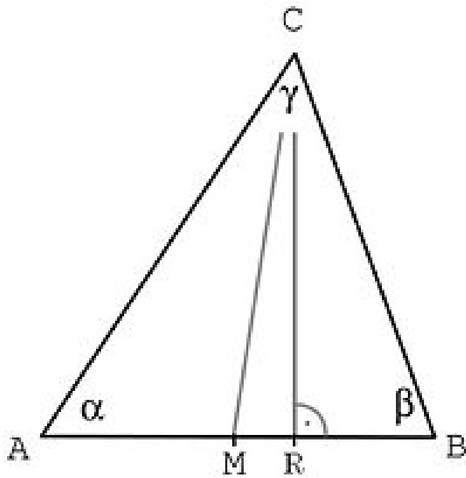


Figure 3. Triangulation example

Now angle C and distances AC and BC can be calculated by law of sines and law of cosines as shown below

$\gamma = 180 - \alpha - \beta$

$$\frac{\sin \alpha}{BC} = \frac{\sin \beta}{AC} = \frac{\sin \gamma}{AB}$$

Cycle avoidance can be implemented by keeping track of the angle of rotation and using X-Y coordinates from the above equations, we can eliminate cycles.

After reaching the destination, the path back to the originating point is calculated using the originating point as (0, 0) and calculating the slope of the line connecting the current position (x, y) and moving in the direction of the origin. While moving towards the origin, the above mentioned wall following, obstacle avoidance strategies can be reused and the slope of the current position with respect to origin gives us the direction to move in order to reach as close as possible to the destination.

References

[1] Youngwoo Ahn, Ja-Ryeong Koo, Animesh Pathak, "CiberMouse motion control scheme for effective path finding," *RTSS CiberMouse 2006.*

[2] Mariano Gomez Plaza, Sebastian Lopez Rodriguez, Sebastisn Sanchez Prieto, and Daniel Meziat Luna, http://www.industrialcontroldesignline.com/howto/201802296.

[3] James Bruce, Manuela Veloso, "Real-Time Multi-Robot Motion Planning With Safe Dynamics".

[4] Maze Solving Algorithms, http://www.astrolog.org/labyrnth/algrithm.htm.

[5] RTSS CiberMouse 2007 Documentation, http://www.ieeta.pt/~lau/web_ciberRTSS07/techdata.htm#docs.

[6] Nageswara S.V. Rao, Srikumar Kareti, Weimin Shi, S. Sitharama Iyengar, "Robot Navigation in Unknown Terrains: Introductory Surver of Non-Heuristic Algorithms," *1993*].

[7] Triangulation, http://en.wikipedia.org/wiki/Triangulation