

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 CyberMouse 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 CyberMouse 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 evolution¹). 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 CyberMouse 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

¹Lamarckian 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.

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 CyberMouse domain is get to the beacon and return back to its starting point 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 CyberMouse goals. These sub-tasks are: explore, hunt, and return task; they are described below.

B. Exploration Task

At the starting point of the CyberMouse simulation, the robot might not sense the beacon, and thus, doesn't 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

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

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 easier 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

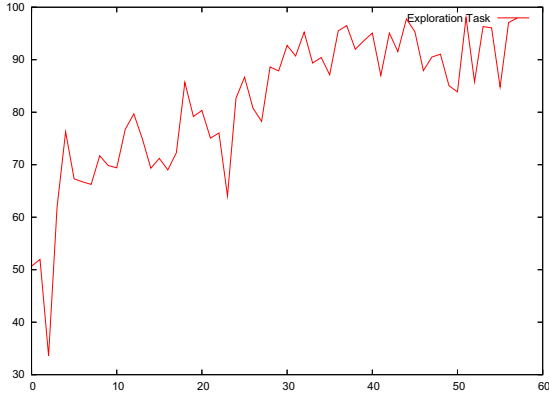


Fig. 1. Exploration task performance

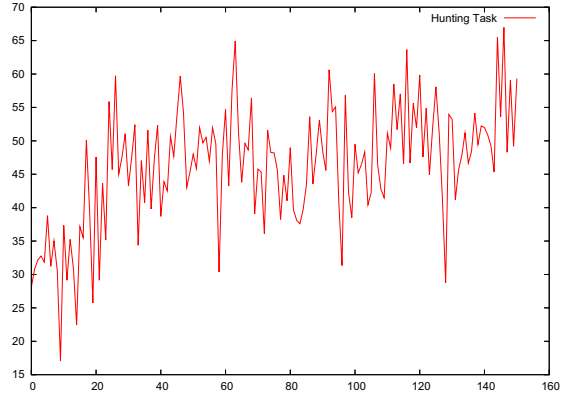


Fig. 2. Hunting task performance

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

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

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 CyberMouse 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 CyberMouse 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 diameters², thus reducing processing time. After this initial sampling, we use only our own estimate of EvoRobert's heading.

B. Sensor Layout

Unlike the standard CyberMouse 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.

²In 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 navigation. 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 robot³. 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 curve⁴, 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

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

for any point $\langle x, y \rangle$ within the arena. $\text{ang}(x, y)$ 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 readings⁵ 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.

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

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

⁵In 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 it⁶.

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 empty⁷.

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.

⁶Note that at a range of one half robot's diameter, the width of the IR sensor beam is 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.

⁷For 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 strictly 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.

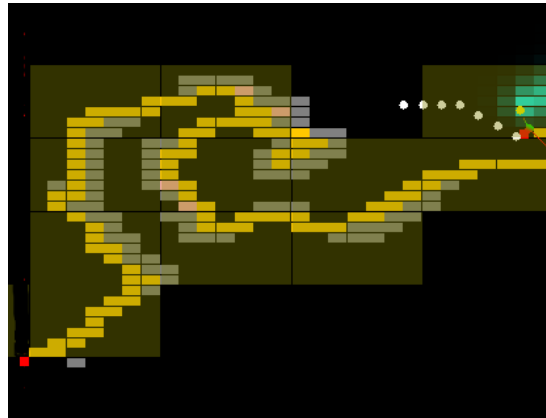


Fig. 3. Java-based visualization tool

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 noticeable 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 CyberMouse*, 2006.
- [Dal05] Robert Daley. *JAGA-SAMUEL A Java Implementation 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.