Genetic Programming to Learn an Agent's Monitoring Strategy

Marc Atkin and **Paul R. Cohen**
Experimental Knowledge Systems Laboratory
Department of Computer Science
University of Massachusetts, Amherst, MA 01003
atkin@cs.umass.edu

Abstract

Many tasks require an agent to monitor its environment, but little is known about appropriate monitoring strategies to use in particular situations. Our approach is to learn good monitoring strategies with a genetic programming algorithm. To this end, we have developed a simple agent programming language in which we represent monitoring strategies as programs that control a simulated robot, and a simulator in which the programs can be evaluated. The effect of different environments and tasks is determined experimentally; changing features of the environment will change which strategies are learned. The correspondence can then be analyzed.

Introduction

It is hard to imagine an agent that does not need to acquire information about its environment, yet the literature on strategies for monitoring the environment is very sparse. The most common monitoring strategies are continuous (i.e., offload monitoring to a dedicated processor) or periodic (see [1] for a survey). However, when monitoring has a cost and the acquired information is noisy, periodic monitoring is provably not optimal. It does not exploit knowledge the agent might have about its environment and the events being monitored, such as approximate probability distributions in time or space for the events. At the same time, while periodic monitoring is often not optimal, it is difficult to say analytically what strategy *is* optimal or even close to optimal, because of the multitude of factors that influence a strategy (e.g., is it better to monitor or take an action? when is the best time to monitor? for what? how should the current state of the environment affect my monitoring strategy? etc.). We therefore use a learning algorithm to find monitoring strategies appropriate to environments and tasks.

Recent work in automated programming has shown it is possible for a genetic algorithm to learn programs; in particular, programs that control a robot's behavior (see [2] for an overview). We use genetic programming to discover monitoring strategies that are difficult or impossible to derive analytically.

Specifically, our algorithm generates programs that move a simulated robot around a grid world. The agent's task—to get as close as possible to an obstacle without touching it, given noisy sensor data—usually can be accomplished more efficiently if it monitors its environment; so in most situations, it learns a monitoring strategy. After we describe the testbed and the way the robot's behavior is represented, we will discuss an experiment in which different monitoring strategies are learned in environments with different features. We will identify a strategy, *proportional reduction*, that is in many cases superior to periodic monitoring. This work is a first step in setting up a general classification scheme that tells us when to use which strategy.

Testbed Description

We have built a simulator in which populations of simulated robots individually operate. Although the simulated world is simple—its only feature is an obstacle, plus dimensions and distance—it allows us to design tasks that require monitoring behavior to be dealt with effectively. By varying aspects of the environment and the task, we can create situations in which we expect different monitoring strategies will be most advantageous. The environment also serves as a testbed in which an agent's monitoring strategy can be evaluated.

Our simulated robots all perform a single task: to reach a goal point in the environment while minimizing their energy expenditure. Our robots have three sensors, one to determine the direction of the goal point, one that reports contact with an obstacle, and a "sonar" that detects the presence of obstacles along the present heading. A robot can move forward, turn left and right, turn itself towards the goal, and use the sonar. These are the basic actuator functions. Monitoring means using the sonar sensor. All these actions require energy, and an energy deduction is also charged for hitting the obstacle. In the experiment described in this paper, we were interested in monitoring strategies that facilitate the robot's task, that is, enable the robot to get as close to the obstacle as possible without touching it. If the sonar were 100% accurate, the robot would monitor only once and move as far as the sonar indicated. In our sce-

This research was supported under the Intelligent Real-Time Problem-Solving Initiative (AFOSR-91-0067) and by NTT Data Communications Systems Corporation.

nario, however, the sonar sensor includes a noise component, normally distributed with variance proportional to the distance measured. A proportionality constant *m* expresses the accuracy of the sensor. A robot must therefore monitor repeatedly to keep the probability of hitting the obstacle low, while at the same time limiting the monitoring cost.

Behavior Representation

The simulated robot is controlled by a program constructed from a language that consists of actuator commands (MOVE, TURNLEFT, TURNRIGHT, GETGOALDIR, MONITOR) and a few control structures (LOOP, IF, etc.). The robot's current knowledge about the environment is held in *environmental variables* (EVAs). They correspond to the sensors available to the robot. Two EVAs are updated automatically: EVA 0—have I reached the goal point? and EVA 1—have I hit an obstacle? Another EVA is required for explicit monitoring: EVA 2—is there an obstacle ahead, and if so, how far away is it?

An agent operating in a real-time environment should be able to respond to external events immediately. To this end, the normal execution of a robot's program can be interrupted when a sensor value (an EVA value) changes. Parts of programs are designated to be interrupt handlers; one interrupt handler is associated with each sensor. If an EVA value changes and the corresponding interrupt is enabled, then the normal program flow is interrupted, and the appropriate handler is executed before returning to the point of interruption.

By disabling and enabling interrupt handlers with explicit commands, a program can easily give some behaviors priority over others, in effect realizing a behavior hierarchy. For example, the main program could be responsible for always moving towards the goal point and monitoring periodically for obstacles, whereas the sonar-associated interrupt handler (EVA 2) would be responsible for turning away from obstacles. It would be executed when an obstacle is detected ahead—it subsumes the move-to-goal behavior. Such a program was indeed found when we ran the algorithm on a scenario in which robots traversed an obstacle-laden environment.

Apart from the choice of actuator and control commands, the interrupt structure was the only architectural feature provided, so as to leave the robot as much freedom as possible in developing its programs.

As it is currently implemented, programs are simply fixed-length linear strings of commands. This distinguishes them from the LISP-style s-expressions used by Koza [3]. While in some ways simpler and more readable, our programs do have the disadvantage of a length restriction. We therefore chose a program length (30 commands) which we knew was sufficient to solve the problem. This restriction also forces the robots to come up with compact solutions, which enhances their readability and, perhaps, generality. (In addition, the explicit LOOP construct makes our representation concise; complex sequences of events need not mean long programs.)

The following example program (Fig. 1) taken from an actual run of the learning algorithm, demonstrates the *proportional reduction* monitoring strategy:

> **Proportional reduction:** Repeatedly estimate the distance remaining to the goal and then move a fixed proportion of the estimated distance.

The proportion determines the probability of overshooting the goal. It is a property of the proportional reduction strategy that this probability remains constant after each monitoring event.

Within the program's outer loop, "LOOP 4 times", the "MONITOR: EVA 2" command is executed repeatedly. Nested within this loop is a second one, "LOOP (EVA 2) times", which takes the current distance to the goal (stored in EVA 2 after monitoring) and loops over its value. Four MOVEs are executed in this loop, each moving .1 distance units. Therefore, the proportionality constant is .4: the robot will move .4 times its estimate of the distance remaining before monitoring again. The command "IF (EVA 2) <= .5 THEN STOP" terminates the program when the obstacle has reached a critical distance, the GETGOALDIR points the robot towards the goal. The NOP commands do nothing; they exist only to make later insertions of new commands easier.

Interrupts are not explicitly enabled in this program, so the interrupt handlers are never called. Only the EVA 2

```
Main program:
    NOP
    NOP
    TURNLEFT
    NOP
    LOOP 4 time(s):
        LOOP 1 time(s):
            GETGOALDIR
            NOP
            NOP
            NOP
            MONITOR: EVA 2
            NOP
            DISABLE: EVA 0
            NOP
            NOP
            MOVE
            IF (EVA 2) <= .5 THEN STOP
            NOP
            LOOP (EVA 2) times:
                LOOP 4 time(s):
                    NOP
                    MOVE
EVA 0 interrupt handler:
EVA 1 interrupt handler:
    IF (EVA 2) <= 4.6 THEN STOP
    TURNLEFT
    NOP
    NOP
EVA 2 interrupt handler:
    DISABLE: EVA 1
    MOVE
    NOP
    WAIT
```

Figure 1: An example program for proportional reduction

handler has any real use in this scenario. One possible use of the EVA 2 handler could be to test if the robot is close enough to the goal instead of doing so in the main loop. Occasionally, programs do actually use the EVA 2 handler in this way. This would mean that the distance is checked immediately after monitoring, since a change in the distance monitored would cause the interrupt handler to be called.

Note that this program is not pure proportional reduction, because there is one MOVE statement in the main loop that gets executed independently of the distance monitored.

The robot's program satisfies two goals at the same time: It controls the robot and describes its strategy. Strategies can sometimes be guessed by looking at a trace of the robot's actual behavior, but it is preferable to read the strategy from the program directly. This is one reason we have shied away from parameter-adjustment algorithms: we want to be able to read the monitoring strategies that our algorithm discovers. As an added bonus, we can also write programs that read the monitoring strategies in the programs. In fact, the data from the experiment described later were collected automatically by a program that reads the programs generated by our algorithm.

The Genetic Algorithm

Programs are learned by a process of simulated evolution known as *genetic programming*. This technique is completely analogous to conventional genetic algorithms (GAs) except that individuals in the population represent programs (see [4] for an excellent introduction to GAs).

The main difficulty is to ensure that the genetic operators, mutation and crossing over, maintain the program's syntactic correctness. We solved this problem by designing the program interpreter in such a way that all individuals that could possibly be generated are legal programs. Range restricted values, such as command numbers and some command parameters, are taken modulo their maximum value, forcing them to be legal. (e.g. the EVA number in particular commands has the legal range of 0 to 2, so taking the EVA number modulo 3 will legalize any value). The same trick applies to other parameters, such as the length of loop bodies, which are taken modulo the program length. Interrupt handlers are simply random pointers into the linear array of program commands; a handler is defined to end when another one starts or when the end of the program is reached. Therefore, changing the start of an interrupt handler by changing the pointer, which can happen during a mutation operation, cannot affect syntactic correctness.

Learning Procedure

At the beginning of each run of the GA, a population of 800 individual programs is randomly initialized; that is, each individual is a random sequence of 30 commands. In each generation, every individual is evaluated by running the program it represents in the simulated environment. Because programs need not terminate, the simulation is bounded by the execution of 1000 program steps.

The individual is then assigned a fitness value based on its average performance on 12 training pairs (start and goal points in the environment) that remain fixed during the learning process. The fitness of an individual determines its chances of survival and reproduction. Fitness consists of two terms, a closeness-to-goal reward, which grows quadratically as the distance between the robot's final position and the goal decreases, minus the robot's energy expenditure on that trial. As the goal is always placed on an obstacle (there are no other obstacles in the map), and the penalty for hitting it is very high, the robot with the highest total fitness is one that goes directly to the goal point, but stops just before hitting it.

Programs from the best 10% of the population are copied unchanged into the new generation. The other 90% are selected based on fitness. We used tournament selection with a tournament size of two: two individuals are selected with replacement and the one with the higher fitness value is chosen. This corresponds to a selection based on the linear ranking of an individual within the population (ref. [5] for discussion). These 90% can be affected by mutation or crossing over: Mutation operator randomly changes commands, command parameters, or pointers to interrupt handlers in the selected individual's program. The mutation rate is initially set to affect every 250th command on average, but it grows, as does the crossing over rate, during long periods in which the best individual does not improve. Crossing over simulates sexual reproduction: two random code segments of equal length are exchanged between two selected individuals. The crossing over rate is initially set to affect every 10th individual on average.

Performance

Although the maximum number of generations was set to 1000, the learning algorithm usually converged within 300 to 800 generations. When there was no improvement in fitness after 150 generations, the algorithm was stopped.

Premature convergence was often a problem. We hoped to counteract it by using tournament selection and dynamically adjusting mutation and crossing over rate to the algorithm's learning rate. Still, there was a considerable variance on the best individual's fitness when the genetic algorithm was run repeatedly. For example, one typical test case would produce individuals whose fitness values ranged from 1502 to 1598 (mean 1548, standard deviation 36.4), corresponding to a set of different monitoring strategies. As the baseline fitness for a program doing nothing was around 1000 for this case, this means a variation of up to 15% in terms of fitness improvement. Although many monitoring strategies have common modules, which is one of the reasons a genetic algorithm should be appropriate to this problem, once an individual had evolved a suitable strategy, it often dominated the rest of the population. We finally chose to run the genetic algorithm several times so as to achieve a better approximation to the best strategy.

The population size of 800 was determined empirically: Although the algorithm's performance (in terms of fitness values achieved) generally increased as the population grew, the effect became weaker after about 500. The deter-

mination of code length (30 commands) was handled similarly.

Experimental Hypotheses

Little is known about optimal monitoring strategies such as proportional reduction (or, for that matter, any other strategies besides periodic monitoring). Psychologists have shown that 14-year-old children use a form of proportional reduction monitoring when solving the "Cupcake Problem," which requires them to take cupcakes out of an oven before they burn, relying on an inaccurate internal clock. Ten-year-olds, however, use periodic monitoring for the same task, because their internal clock is less accurate or because they simply have not developed the proportional reduction strategy [6]. Cohen demonstrated that proportional reduction is optimal if the cost of missing a deadline does not depend on how badly the deadline is missed, but dynamic programming solutions to a more general problem show that proportional reduction is only a close approximation to the optimal monitoring strategy [7].

We wanted to see if and when proportional reduction would emerge in our simulated robots. We suspected that several features of the environment and the robot would influence monitoring strategy. The error in the "sonar" data was of particular interest, as the proportional reduction constant should depend directly on the sonar noise. A lot of noise should result in a more cautious program (with a low constant) that monitors after moving shorter distances. We hypothesized that when sonar data are extremely noisy, the proportional reduction strategy would show no advantage over periodic monitoring, as the data no longer contain any useful information. (It is easy to show mathematically that periodic monitoring is optimal when no information is given about the occurrence of an event.)

Theoretically, proportional reduction should show more of an advantage as the distance between the robot's starting position and the goal grows. Imagine a periodic monitoring strategy that monitors every five distance units; this strategy will monitor six times if the distance between the start state and the goal is 32. Now imagine a proportional reduction strategy with .5 as its constant. On average it will monitor once to start and then after traveling 16 units, then 8, then 4, then 2, then one. But if the distance between the start and goal states is doubled, the proportional reduction strategy will monitor one additional time, while the periodic strategy will monitor twice as much. This is why the advantage of proportional reduction should be more apparent when the start and goal states are more separated. (Note that this distance cannot be a specific length, but must be a range, otherwise the genetic algorithm will over adapt to move exactly the required distance, completely circumventing any need to monitor.)

To summarize our hypotheses:

1. If monitoring error, m, is 0, a robot should monitor once only.
2. The PR constant becomes smaller as m increases.
3. Periodic monitoring becomes more likely as m increases and path length decreases.

Experiment Results

We ran the genetic algorithm on five levels of m, the monitoring error (0, 0.5, 1.0, 2.0, 3.0) and three ranges of path length (1–4, 5–10, 12–18). To increase the chances of getting a good strategy in each of these cases, we repeated the process 10 times with different random number seeds for the genetic algorithm, taking the overall best program (in terms of fitness) from these runs as our final result.

This program was then re-tested on 36 start-goal pairs in each specified length range. Four measures were averaged over the 36 trials to evaluate the program's performance and strategy:

1. fitness
2. total cost of monitoring
3. percentage of distance traversed while using proportional reduction
4. proportional reduction constant

The first three measures were calculated by an evaluation function directly from the program's execution trace. The fourth was determined manually by looking at the program; but it should not be hard to automate also. Note that measure 2 basically tells us how many times the agent monitored, as monitoring has a fixed cost.

Figure 2 shows the average fitness values for the final program on the 36 test pairs. For the most part, fitness values decrease monotonically as m and path length increase, with the exception of the m=3.0, path length=1–4 situation. Both these effects were predicted: Longer path lengths mean higher energy consumption, which lowers fitness, and higher values of m mean more elaborate and energy-consuming monitoring strategies. Note however that path length has a much greater effect than monitoring error. Apparently the genetic algorithm copes quite well with finding good monitoring strategies. An ANOVA shows a clear effect of path length and monitoring error on fitness (for both main effects $p < .0001$), but no interaction effect.

The exception mentioned above is in fact actually due to the values for the m=1.0 and m=2.0 (range 1-4) cases being too low. In these cases, the robot hit the obstacle once or twice on the 36 test pairs, whereas it didn't on the 12 training pairs. Obviously the solution to this problem is to

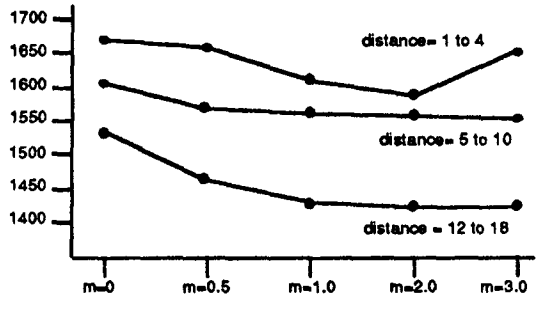

Figure 2: A plot of cell means for a two-way ANOVA, monitoring error (m) by distance. The dependent measure is the *fitness* of the best individual to evolve in 12 trials, re-tested on 36 trials.

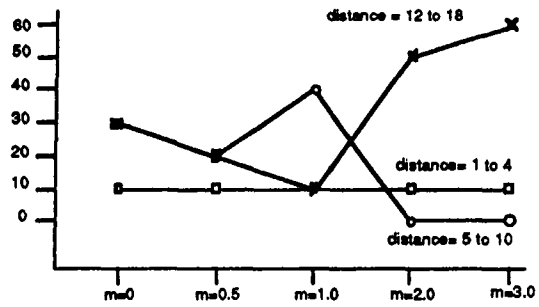

Figure 3: Average monitoring cost for 36 test cases for each of three levels of distance and four levels of monitoring error.

train on more pairs, but it is a trade-off between better results and run time of the algorithm.

Figure 3 shows the mean monitoring costs. We had expected to see monitoring cost increase as path length and especially monitoring error increases. This prediction is only partly verified. The expected pattern is distorted because when the monitoring error gets too high, the genetic algorithm does not come up with a monitoring strategy at all, but instead tries to find a different solution such as deliberately moving past the goal point (to the left or right) for a distance that is approximately the average of the path lengths of its training set. Table 1 gives an overview of the best individual's monitoring strategy for each test case. As you can see, in cases m=3.0, path length 5-10; and m=2.0, path length 5-10, no monitoring strategy was found. Sometimes (case m=1.0, path length 1-4 and path length 12-18), it will combine the strategy of moving past the obstacle with proportional reduction, ensuring that if proportional reduction fails, it will still not touch it.

length	$m = 0.0$	$m = 0.5$	$m = 1.0$	$m = 2.0$	$m = 3.0$
1-4	PR^2 (**)	PR (.7)	PR^2 (*) (**)	PR (.5) (**)	check (*) (**)
5-10	PR^2	DPR	PR (.4)	-- (*)	-- (*)
12-18	DPR	PR (.7)	DPR (*) (**)	DPR	DPR

Key:

PR (c):	proportional reduction strategy with proportional reduction constant
PR^2:	moves a proportion of the squared distance remaining
DPR:	disproportionate reduction: moves a proportion of distance remaining plus a constant distance.
check:	stops if obstacle is visible after monitoring
--:	no monitoring strategy
(*)	deliberately moves past goal to one side
(**)	strategy is only capable of monitoring once

Table 1: The best individual's monitoring strategy

Interestingly, no program monitors more than once when the path length is very small. We had expected the "monitor only once" strategy only in the absence of sensor noise, but apparently, over such short distances, the extra effort involved in coming extremely close to the goal is not worth the energy necessary to achieve it. Another surprise is that for longer path lengths and no sensor error, programs still monitor more than once. It is hard to imagine how this could be advantageous.

Some form of proportional reduction was found in nearly all situations, as Figure 4 shows. In fact, in all but one of the cases where a monitoring strategy was found, proportional reduction was used. Our hypothesis that periodic monitoring would become dominant for high values of monitoring error was not confirmed. It is important to mention, however, that periodic monitoring was indeed discovered by the algorithm in several high-monitoring-error situations, but its fitness was slightly lower than proportional reduction. The one clear exception is the m=3.0, range=1–4 case. Here, the robot moves a certain distance, then monitors for the obstacle. If the obstacle is visible ahead, it stops; otherwise it moves a fixed distance further, curving around the obstacle. This is notable because it is not a proportional reduction strategy, the sensor data are too inaccurate: the sensor is used only to detect the presence or absence of the obstacle.

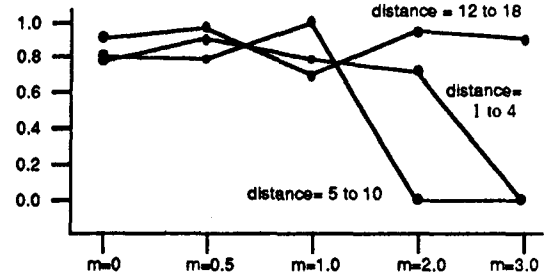

Figure 4: Percentage of the distance between the starting state and the goal in which the robot uses the proportional reduction strategy.

The proportional reduction strategy did not usually manifest itself in its pure form. Frequently, the robot would move a short distance before beginning to monitor with the proportional reduction strategy. This is not surprising because in many of the trials, the robot was guaranteed a minimum distance from the goal by the lower bound on the path length. Sometimes after monitoring, in addition to moving a distance proportional to the remaining distance, the robot moves a fixed distance as well. We call this "disproportionate reduction", as the robot is in effect moving a larger proportion of the distance remaining to the goal as it gets closer to the goal. In pure proportional reduction, this proportion remains constant throughout the strategy. The example program presented in a previous section implemented this behavior. Note that the fixed additional distance was usually small with respect to the distance moved via proportional reduction. It is therefore very possible that the extra MOVE commands that cause it are simply a relict of the evolutionary process: They did not cause much harm, and were therefore not removed in later generations.

Another variant of the original proportional reduction scheme is something we termed "squared proportional reduction". In this strategy, the robot monitors repeatedly, and moves a proportion of the *squared* distance remaining. It is realized by nesting two EVA 2-loops within each other after the MONITOR command:

```
MONITOR: EVA 2
LOOP (EVA 2) times:
        LOOP (EVA 2) times:
                MOVE
```

As shown in Table 1, this strategy was learned only occasionally. In all the situations where it occurs, the agent only monitored once, or m=0. In the first case, the proportionality constant (which corresponds to the number of MOVEs within the doubly nested loop) can be adjusted to provide a good estimate of the distance remaining given the specific path lengths. As it is expressed here, we do not see squared proportional reduction as a distinct strategy. The second case also seems to be a form of over-adaptation: the constant is again adjusted to fit the particular range of path lengths in that situation. Perhaps with squared proportional reduction and no monitoring error, the robot can reach the goal faster than with proportional reduction alone.

Our second hypothesis was that the proportionality constant should decrease as m grows. The proportionality constant is the proportion of the robot's estimated distance to the goal that the robot actually moves before monitoring again. Of course this constant is only really defined in cases of unmodified proportional reduction, which limits our available data points. But the few applicable cases do show the predicted effect (see Table 1). The range of the constant decreases from .7 in two of the m=0.5 cases to .4 in the second m=1.0 case. The constant of .5 in the m=2.0, path length = 1-4 case should not be weighed too strongly as the robot only monitors once here. Even so, it is still close to the previous .4 value.

The evidence for our other hypotheses, that the robot will monitor only once when there is no monitoring error, and that periodic monitoring will surpass proportional reduction as m and path length increase, is not so clear-cut. The proportional reduction strategy did emerge, but frequently not in its pure form. Periodic monitoring never beat proportional reduction, although it did come close in the m=2.0 cases. We are lead to believe that periodic monitoring will only be advantageous when virually no distance information is given. We are currently attempting to show this mathematically.

Two interesting new monitoring strategies emerged-ones we hadn't foreseen: disproportionate reduction and squared proportional reduction. They seem to be quite complementary in their relationship to unmodified proportional reduction. The first moves relatively greater distances as it approaches the goal, the second moves more quickly while it is still further away. It is an open question if either of these strategies is really doing some useful, beyond what pure proportional reduction can do. But the higher rate of occurrence for the disproportionate strategy as path length increases might be worth looking into.

Another phenomenon was also quite surprising: When the monitoring error becomes too large, programs emerge that try to do without monitoring. It was interesting that the non-monitoring program in the m=2.0, path length=5-10 case had all the constructs to do proportional reduction, except the actual (and very expensive) monitoring command. It seems that these particular individuals had evolved beyond the proportional reduction strategy, choosing instead a scheme of passing by the goal point and not monitoring at all.

Discussion and Future Work

Genetic programming might turn out to be a very robust way of finding solutions to hard behavioral control problems such as monitoring. It takes particular advantage of the *modularity* of programs, combining them in different ways to find novel solutions. We are pursuing the idea of learning to construct programs from clearly identified sub-blocks, and evaluating these blocks in terms of their global usefulness.

Genetic algorithms are however challenging in that they do not guarantee optimal results. In our experiment, it was often difficult to determine whether a new strategy is actually an interesting result, or merely sub-optimal. As finding the best possible solution is the task at hand, there is no absolute by which to judge strategies. Testing strategies in other domains on similar problems, thus compensating for over-fitting, could provide a more neutral evaluation.

As a follow-up experiment to this one, it would be interesting to see how well robots can do if they don't have the constructs to implement proportional reduction. Or when given the freedom to realize more sophisticated strategies, will proportional reduction still emerge? In what form? For what other situations are the discovered strategies useful? What features of the environment determine their success?

Clearly the work presented here is only a first step in achieving an understanding to how the environment determines what is a good monitoring strategy. To do this topic any justice, many further experiments are necessary. The ultimate goal must be to find general rules of when to use a specific strategy; rules that are general enough to be applicable across domains.

References

[1] Hansen, E.A. & Cohen, P.R. 1993. Monitoring plan execution: A survey. In preparation for *AI Magazine*.

[2] Koza, J.R., 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection and Genetics*. MIT Press, Cambridge, MA.

[3] Koza, J.R. & Rice, J.P., 1992. Automatic Programming of Robots using Genetic Programming. *AAAI-92*, Pp. 194-207

[4] Goldberg, D.E., 1989. *Genetic Algorithms in Search, Optimization & Machine Learning*. Addison-Wesley.

[5] Goldberg, D.E. & Kalyanmoy, D., 1991. A Comparative Analysis of Selection Schemes Used in Genetic Algorithms, in *Foundation of Genetic Algorithms* (Gregory J.E. Rawlins ed.). Morgan Kaufman.

[6] Ceci, S.J. & Bronfenbrenner, U., 1985. "Don't forget to take the cupcakes out of the oven": Prospective memory, strategic time-monitoring, and context. *Child Development*, Vol. 56, Pp. 152-164.

[7] Hansen, E.A., 1992. Note on monitoring cupcakes. *EKSL Memo #22*. Experimental Knowledge Systems Laboratory, Computer Science Dept., Univ. of Massachusetts, Amherst.