In *Proceedings of ICEC '94, World Congress on Computational Intelligence.*

Learning Monitoring Strategies: A Difficult Genetic Programming Application

Marc S. Atkin and Paul R. Cohen

Computer Science Technical Report 94-52

Experimental Knowledge Systems Laboratory
Department of Computer Science, Box 34610
Lederle Graduate Research Center
University of Massachusetts
Amherst, MA 01003-4610

Abstract

Finding optimal or at least good monitoring strategies is an important consideration when designing an agent. We have applied genetic programming to this task, with mixed results. Since the agent control language was kept purposefully general, the set of monitoring strategies constitutes only a small part of the overall space of possible behaviors. Because of this, it was often difficult for the genetic algorithm to evolve them, even though their performance was superior. These results raise questions as to how easy it will be for genetic programming to scale up as the areas it is applied to become more complex.

This research is supported by ARPA/Rome Laboratory under contract #'s F30602-91-C-0076 and F30602-93-0100.

1 Introduction

Every agent needs to monitor its environment. The strategy for doing this must weigh the cost of monitoring against the cost of having inaccurate information about one's current situation. For the past two years, we have been working on different systems that—when given the agent specification, its task, and the environment in which it operates—learn these strategies automatically. In all systems, the learning scheme used was some form of evolutionary computation. In all but one, it was genetic programming.

Our research goal was to use these systems to discover and categorize monitoring strategies. We wanted to generate a lot of different *scenarios* (sets of tasks and environments), run the genetic algorithm on each of them, and record the monitoring strategy that emerged in a particular situation, the implicit assumption being that is was fairly well suited to the given circumstances. Then, we would group the monitoring strategies and classify the groups by general features of the scenarios, ending up with a *monitoring strategy taxonomy.*

Unfortunately, getting the genetic algorithm to this point proved to be a very laborious undertaking. Somewhat surprisingly, the problem was only secondarily one of tuning the genetic algorithm to the particular task—it would generally perform quite well and produce some kind of sensible solution to the problem we gave it. It was primarily one of steering the genetic algorithm towards solutions *we* were interested in, namely monitoring strategies. We believe that the difficulties we encountered might be an indicator of a more fundamental problem relating to a general weakness of genetic programming: Can they deal effectively with very large search spaces? This paper will give a description of the problems we had, how we attempted to tackle them, and why we think they arose.

2 A Family of Systems that Learn to Monitor

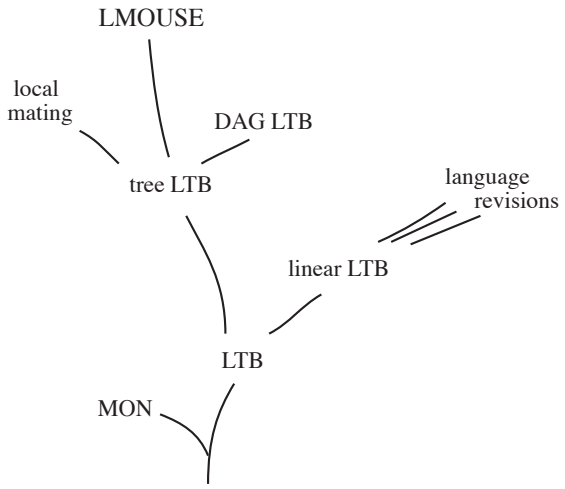

Figure 1: A Family Tree of Monitoring Systems

Our first system, creatively named "MON" (for monitoring), was a pure genetic algorithm. Given a parameterized set of functions describing when to monitor next (e.g. $a\log(t)+b$), it would find values for the parameters (e.g. a and b) that minimized expected cost. This cost consisted of two parts: A cost for monitoring, i.e. a fixed cost charged for every time the environment is probed, and a cost that is determined by the task, the *task penalty.*

What do these tasks look like? Here are two examples from the time domain: In the so-called *cupcake problem* [4], the task is to take cupcakes out of an oven as close to a predetermined time d in the future as possible. This problem would be simple if one had an internal clock that was completely accurate. In this scenario, however, it is not. But it can be recalibrated by taking a monitoring action, i.e. looking at a wall clock. The task penalty expresses how far in time from d the cupcakes were actually removed. Since monitoring has a cost associated with it, this time differential cannot be made arbitrarily small without occurring a very high monitoring cost. MON was able to show empirically (see [1]) that periodic monitoring is not the optimal strategy for this task. We now have verification that a strategy called *interval reduction* (monitor more frequently as d gets closer) is in fact the best monitoring strategy (see [7]).

In another task, which corresponds to monitoring for an event which we will assume occurs at a fixed rate at any given time, say fire breaking out in a forest, the task associated penalty is a function of the time elapsed between the event and it being detected by the monitoring action. It it easy to prove that for this problem, periodic monitoring is the optimal strategy. Periodic monitoring was also discovered by MON.

Spawned by the positive experiences with MON, we decided to expand the idea of learning a monitoring strategy with a genetic algorithm to a more general class of problems. A monitoring strategy depends not only on the temporal placement of monitoring actions, but also on what the agent is doing when it is not monitoring. Our second system, "Learning to Behave" (LTB), learned not only when to monitor, but also how to accomplish a given task. It learned *behavior*.

Behavior was represented in the form of a program which controlled a robot in a fairly simple, simulated, two-dimensional world. The programming language itself consisted of basic effector commands such as "MONITOR, "MOVE", or "TURNRIGHT", and some control structures such as loops and later conditionals. Now, genetic programming was used as the learning mechanism. The task of the learning algorithm had become somewhat more complex: it had to evolve complete programs, guided only by a *fitness value* expressing how well on average this particular program did in the simulated environment. Apart from giving the robot enough basic commands to solve its given task (usually one of obstacle avoidance), we supplied it with a mechanism to respond immediately to external events, something which is very important for a real-time agent. Each robot was equipped with a number of sensors. Associated with each sensor is a piece of code called an *interrupt handler*. When a sensor value changes, normal program flow is interrupted, and the corresponding handler is executed in the manner of a sub-program. Interrupts can be disabled or enabled via explicit commands, making it possible for the robot to prioritize handler execution. An example of interrupt handler use will be given in the next section.

LTB went through a number of revisions, as can be seen in figure 1. In the initial representation, programs were linear lists of commands. They could contain structures such as LOOP's and IF's, but their total length was limited. The language was changed several times. Originally, the representation was much closer to assembly language than it is now. All control structures were implemented as GOTO's, but this was changed later to make programs more readable. Some commands were added, such as a construct that loops a variable number of times. We did a number of experiments using monitoring to facilitate obstacle avoidance, using a very basic genetic programming model. Crossing over simply swapped two chunks of code between two individuals, mutation changed one command into another. Since the language was relatively simple, only a few constraints were needed to keep languages legal, such as ensuring via a modulo operator that command values and parameters (e.g sensor numbers) were kept within their legal ranges. Selection was scaled roulette wheel as described by Goldberg [5, pp. 76-79], later tournament selection with a tournament size of two was used to counter premature convergence (see [6] for a comparison). LTB did indeed find good monitoring strategies for the obstacle avoidance tasks.

The linear representation seemed limiting. Crossing over had no semantics. There was no notion of "behavior modules", useful blocks of code that were responsible for a specific kind of sub-behavior, as there is in a tree representation used by Koza [9, 10] and many others in the field. We rewrote LTB to operate on this representation, using parse trees to represent the main program and the interrupt handlers. We were hoping that it would improve performance. As we will see in the next section, it surprisingly did not.

We wanted to make our behavior description language as general as possible, since our monitoring strategy taxonomy required a large number of scenarios to have any weight, and the agents should be well equipped to solve most of tasks we required them to. But as we added constructs to LTB, such as conditionals, compare statements, and more interrupt handlers, monitoring strategies emerged less and less frequently, even though the fitness functions were designed to favor them.

We did number of experiments to alleviate these problems, such as determining which settings of parameters in the fitness function (such as the cost for hitting an obstacle, monitoring, etc.) maximized the fitness difference between monitoring and non-monitoring programs. We also tried using local mating (also known as "fine grain parallelism"; for a description see [3]) to fight the premature convergence that we thought was producing the suboptimal results.

These difficulties with LTB led us away from learning a complete behavior to solve a given task, and back to the problem we were really interested in: monitoring strategies. LMOUSE ("Learning to Monitor Using Simulated Evolution") was a system that focused on this aspect. LMOUSE learns functions (explicitly represented in the form of evaluation trees, unlike in MON), that are evaluated to determine when to monitor next. There is one function associated with each sensor. The tree's terminals are either sensor values or constants, and the interior nodes one of the four basic mathematical operators, a comparison, or a conditional (a conditional evaluates its first argument; if it is non-zero, it returns the value of the second argument, otherwise the third). The program controlling the agent is provided by the experimenter. It can be arbitrarily complex, and may very well contain references to sensor values—values that the genetic algorithm must determine when to update. One would think that we have simplified the problem, since the system is no longer responsible for solving the complete task, only for interleaving monitoring events into its controlling program. But as we shall see, this was still a very hard problem.

3 Monitoring in Obstacle Avoidance Tasks

Most of the tasks we presented our systems were obstacle avoidance tasks[1]. The *environment* was a two-dimensional world divided into 800 square fields. Each field had a unique "terrain" attribute, for example "obstacle" (robot could not enter this field) or "grass" (no hindrance to the robot's movement). Even though the terrain was discretized, the robot could still move continuously within the map. The obstacles were typically arranged in the center of the world map to form a large rectangular barrier.

The robot's task was to move from a start point to goal point without hitting the obstacle. The robot was rewarded for getting close to the goal, and penalized for every time it attempted to move onto an obstacle field. These were the main terms in the robot's fitness function, in addition the robot was charged a small amount for every effector action it took, plus a monitoring cost for each time it actively used a sensor. The fitness was the average of the robot's performance on ten random yet fixed start-goal pairs.

[1]The notable exception being a comprehensive experiment we did to compare a particular type of interval reduction, *proportional reduction* (the time between monitoring events is reduced proportionally as the deadline approaches) to the strategy of periodic monitoring [2].

The robot had three sensors, one that told it if it had reached the goal, one that told it if it had hit an obstacle, and one that could detect an obstacle ahead. The first two were mainly provided so that the robot could complete its task; their values were automatically updated. The third one, the "sonar", was a lot more interesting. It had to be actively queried via a MONITOR command to have it update its value. To make the sensing task more difficult (and realistic), a noise term could be added to the value it returned, simulating sensor noise. The sonar was the only sensor that could be used to prevent a collision with an obstacle, so it was the one we hoped would be used in a monitoring strategy.

And indeed, it was. After tuning the genetic algorithm, in particular, adding a scheme that dynamically increased mutation and crossing over rate if there were long periods during which the best individual in the population showed no improvement, once in a while the genetic algorithm would find a monitoring strategy. And because its fitness was generally higher than the fitness of the other strategies found which scrambled around the obstacle on their way to the goal, we were led to believe that the sub-optimal results we got could be avoided by more careful adjustment of the learning algorithm. Further experiments to determine good values for other parameters such as population size led to an increase in the percentage of times linear LTB would find a monitoring strategy to about 50%. Figure 2 shows the best program the system came up with for a program size of 20 commands.

```
Main program:
 TURNTOGOAL
 ENABLE: reached_goal
 NOP
 NOP
 ENABLE: object_distance
 LOOP infinitely:
   LOOP 7 time(s):
     MOVE
     MOVE
   ENABLE: hit_object
   TURNTOGOAL
   MONITOR: object_distance
*reached_goal* interrupt handler:
 STOP
 STOP
 NOP
*hit_object* interrupt handler:
 WAIT
 STOP
 NOP
*object_distance* interrupt handler:
 TURNRIGHT
 MONITOR: object_distance
```

Figure 2: A monitoring strategy for obstacle avoidance behavior (linear LTB, population 800, 2500 generations).

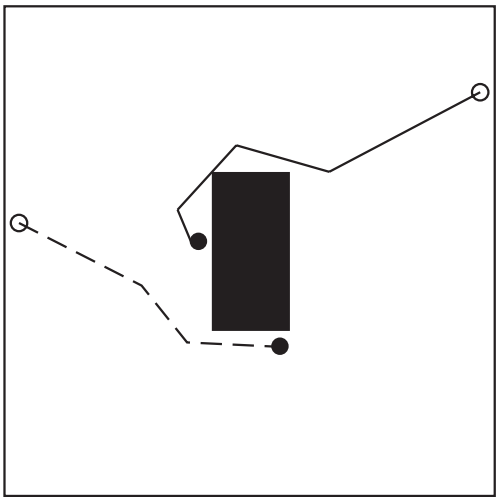

Figure 3: An execution trace generated by the program in figure 2 for two of the ten start-goal pairs. Hollow circles denote starting fields, full circles goal fields.

Its structure is quite simple: After enabling the interrupt handlers corresponding to the "reached goal" and "object distance" sensors and turning itself towards the goal with "TURNTOGOAL", the program goes into an infinite loop. Within this loop, it periodically moves 14 times, which corresponds to 2.8 field widths, before monitoring for an obstacle ahead (via "MONITOR: object_distance", which sets the object_distance sensor to be the distance to the obstacle—or zero, if there is none). Since the "object distance" interrupt handler had been previously enabled, when an obstacle is detected, this interrupt handler will be called. In it, the robot turns right by 22.5 degrees, and then monitors for the obstacle again. If it is still visible, the interrupt handler will be called recursively, turning the robot further. This will continue until the obstacle is no longer visible, then the agent will move forward for 2.8 fields in its current direction before turning itself towards the goal again and rechecking for the presence of an obstacle. When the goal point is reached, the "reached goal" handler will be invoked, which ends the trial with the "STOP" command. Although the "hit object" handler is enabled, it never gets called. Figure 3 shows the path of the robot for two of the start-goal pairs. The changes in direction away from the obstacle denote the points where monitoring took place.

Surprisingly, the version of LTB which used the tree representation only found a monitoring strategy on about 30% of the runs of the genetic algorithm. So the more flexible representation seemed to actually be hurting performance. It would be nice to see a comprehensive comparison between these two forms of representation in the future.

```
Main program:
  LOOP 24 time(s):
    MOVE
    ENABLE: hit_object
    MOVE
    MOVE
  COMPARE 47 to (terrain_type)
  LOOP 21 time(s):
    TURNTOGOAL
    ENABLE: goal_direction
    MOVE
    MOVE
    LOOP 565 time(s):
      MOVE
  TURNTOGOAL
  MOVE
  IF <smaller>
    TURNTOGOAL
    ENABLE: goal_direction
  ELSE
    SHOOT
    MOVE
  MONITOR: goal_direction
*robot_direction* interrupt handler:
  MOVE
  SHOOT
*robot_speed* interrupt handler:
  MONITOR: terrain_type
*reached_goal* interrupt handler:
  COMPARE 132 to (object_distance)
*goal_direction* interrupt handler:
  EXIT
*hit_object* interrupt handler:
  SHOOT
  SHOOT
  SHOOT
  IF <not equal>
    SHOOT
  ELSE
    MOVE
    MOVE
    MOVE
  SHOOT
*object_distance* interrupt handler:
  ENABLE: object_distance
*terrain_type* interrupt handler:
  ENABLE: robot_speed
*obstacle_density* interrupt handler:
  TURNLEFT
```

Figure 4: The best found strategy for obstacle avoidance with added language constructs and interrupt handlers (tree LTB, population 800, generations 643).

In our attempt to apply LTB to a greater range of monitoring tasks, we added a number of constructs to the basic command set, including an explicit IF-construct (previously, all conditional actions had to be handled via interrupt handlers), COMPARE-construct, several new sensors with corresponding handlers, and a SHOOT command, which destroyed the first obstacle within a certain distance in front of the robot. To balance its power, using SHOOT was made fairly expensive. The motivation for SHOOT was the following: In previous avoidance tasks, a lot of effort not only went into detecting an obstacle, but also finding a good strategy to circumvent it once it had been detected. We hoped to make things easier for the robot by introducing the SHOOT command, since we were mainly interested in monitoring anyway. A robot could now destroy a detected obstacle immediately, without the need for a fancy avoidance behavior. Figure 4 shows the best output of tree LTB (over 10 separate runs of the system) for the same obstacle avoidance task.

Instead of using its object distance sensor, the program uses SHOOT to clear a path through the obstacle. The "hit object" handler is called the first time the robot encounters the obstacle. In it, "not equal" will be true, and SHOOT will be executed a total of four times. Since SHOOT was so expensive, the program minimizes its usage by approaching the obstacle orthogonally, and only later turning completely towards the goal. While this solution is certainly ingenious, it is be no means the best. Using monitoring would have resulted in a higher fitness. The obvious first thought was that SHOOT was somehow distorting the solution. But removing it didn't help: the robot now simply stopped when it hit the obstacle, and made no attempt to even scramble around it.

One such case of difficulty in finding a monitoring strategy might be explicable with a poorly designed simulator, fitness function, or genetic algorithm. But we experienced the same difficulties on the wide range of problems and environments we had selected for the strategy taxonomy experiment (avoidance of uniformly distributed obstacles, gradient distributed obstacles, exploration tasks under two different agent architectures, monitoring for "predictor" events, etc.). The fact that the very same task that had been solved fairly easily before (figure 2) now seemed intractable, spoke for itself. The algorithm's difficulties had to have something to do with increased number of program constructs and interrupt handlers.

We had realized that adding constructs would increase the size of the search space, but this did not originally worry us. The search space size had increased, but hadn't also the number of "good" solutions? Apparently, the former grows a lot faster than the latter. If this phenomenon applies only to monitoring strategies, things wouldn't be so bad. But this seems implausible: For most reasonably complex tasks, the set of optimal solutions should be comprised of only a small number of distinct strategies. There might be many variations on a theme, but the number of basic, generalized, strategies should stay small. Any particular strategy will use a certain number of sensor values, and a certain configuration of program constructs. If this is the case, adding new sensors or constructs will decrease the size of the set of good solutions considerably, and we think that is exactly the explanation for the problems we had.

Even LMOUSE, which was designed in such a way that every strategy it came up with was a monitoring strategy, needed a lot of tweaking before it came up with a strategy that made any kind of sense. As is typical with genetic programming, the system would exploit some (unintended) property of the simulator to achieve a fairly decent result. A followup experiment showed why LMOUSE was having so much trouble: we implemented a search that sampled 500000 random programs before terminating. Of these 500000, the best one found was one that did nothing—it terminated the trial immediately. It seems that useful monitoring strategies are also very rare in the search space of this system, and it is impressive that LMOUSE could find one at all.

It could be objected that we cannot expect a genetic algorithm to do well in search spaces where good or desired solutions are so sparse. This is certainly correct, and by no means the fault of the

algorithm—it is doubtful that any other weak search method would do any better. But this is not the issue. The issue is that we will need expressive behavior languages to be able to solve more complex problems. We will not necessarily know how to constrain the command and sensor sets a priori. If the above argument holds, these languages might cause very sparse search spaces to become a lot more frequent, and methods must be found make them tractable.

4 Can Genetic Programming Scale Up?

The work we have presented is work in progress. We were side-tracked from our goal of a monitoring strategy taxonomy into explaining why our systems had so much trouble discovering monitoring strategies. We plan to continue our investigation of this area, and establish a quantative measure of the relationship between search space size and a genetic programming systems' performance, at least for the task at hand.

What happens though if we or others discover search space size is linked inversely to solution density, which in turn decreases performance? Can a genetic algorithm of any flavor be used to cope with the type of search spaces we were forced to deal with? Can genetic programming techniques be scaled up to languages that contain hundreds of commands and variety of complex control constructs? As is known from theoretical work on genetic algorithms (e.g. [13]), as the complexity of the encoding language increases, larger populations are required to get the same allele coverage. But increasing the population size to incorporate a proportion of the very rare solutions will only work up to a point. Today's fastest machines often take days or weeks to solve problems of any consequence, and even parallel architectures will not let populations increase indefinitely. Amplifying the problem is that the fitness for many genetic programming applications is determined through simulation, a very time-consuming activity. Manually having to adapt a general purpose language to the specific task that needs to be solved isn't a satisfying solution, either.

A promising way out of this dilemma might be *modularization*. Some fairly recent work [11] focuses on ways of having genetic programming algorithms find potentially useful sub-behaviors and then building more complex behaviors from them. In effect, this would reduce the size of the search space by reducing the number of command combinations; instead of combining basic effector actions, the algorithm is working with pre-formed behavior modules. It also makes the output of the algorithm a lot easier to understand. Another possibility might be to insert already proven strategies into the initial population [12].

While it is certainly too early to draw too many far-reaching conclusions, the nature of search spaces is certainly something worth keeping an eye on, or others might be as unpleasantly surprised as we were. Genetic programming seems to be a very powerful technique, but if it is to have as much impact as its proponents hope it will, it will have to face the problem of scaling up sooner or later.

Acknowledgments

This research is supported by ARPA/Rome Laboratory under contract #'s F30602-91-C-0076 and F30602-93-C-0100. The US government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon.

References

[1] Atkin, M.S., 1991. Research Summary: Using a Genetic Algorithm to Monitor Cupcakes, *EKSL Memo #24*, Experimental Knowledge Systems Laboratory, University of Massachusetts, Amherst.

[2] Atkin, M. & Cohen, P.R., 1993. Genetic Programming to Learn an Agent's Monitoring Strategy, *Proceedings of the AAAI 93 Workshop on Learning Action Models*, Pp. 36-41.

[3] Baluja, S., 1993. The Evolution of Genetic Algorithms: Towards Massive Parallelism, *Proceeding of the Tenth International Conference on Machine Learning*, Morgan Kaufman, San Mateo, CA., Pp. 1-8.

[4] 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.

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

[6] Goldberg, D.E. & Kalyanmoy, D., 1991. A Comparative Analysis of Selection Schemes Used in Genetic ALgorithms, in *Foundations of Genetic Algorithms* (Gregory J.E. Rawlins ed.). Morgan Kaufman, Pp. 69-93.

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

[8] Hansen, E.A. & Cohen, P.R., 1994. Monitoring the execution of robot plans: A survey. Submitted to *AI Magazine.*

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

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

[11] Koza, J.R., 1993. Simultaneous Discovery of Reusable Detectors and Subroutines Using Genetic Programming. *Proceedings of the Fifth International Conference on Genetic Algorithms*, Morgan Kaufman, San Mateo, CA., Pp. 295-302

[12] Ramsey, C.L. & Grefenstette, J.J., 1993. Case-Based Initialization of Genetic Algorithms. *Proceedings of the Fifth International Conference on Genetic Algorithms*, Morgan Kaufman, San Mateo, CA., Pp. 84-91

[13] Tate, M.T., 1993. Expected Allele Coverage and the Role of Mutation in Genetic Algorithms, *Proceedings of the Fifth International Conference on Genetic Algorithms*, Morgan Kaufman, San Mateo, CA., Pp. 31-37