Evolving Complex Visual Behaviours using Genetic Programming and Shaping*

Simon Perkins s.perkins@ed.ac.uk

Gillian Hayes gmh@dai.ed.ac.uk

Department of Artificial Intelligence, University of Edinburgh,
5 Forrest Hill, Edinburgh, Scotland

**Presented at the 7th European Workshop on Learning Robots, Edinburgh, July 20th 1998*

Abstract. Learned or evolved robot controllers are often criticized for being unable to compete with hand-designed controllers. Hierarchical structuring of the controller coupled with staged learning can enhance the capabilities of learned controllers, but is subject to the criticism that the human designer is in effect doing all the hard work, and hence that this approach will not scale up. We suggest imposing an explicit barrier between the human designer and the internal workings of the controller, limiting the human to using external methods for influencing the learning process that might be more scalable. Such methods are similar to what is called 'shaping' in animal training. In this paper we first introduce a simple but powerful GP-like learning architecture. We then enhance this by incorporating it initially into a rule-based system, and then into a multi-agent system. The utility of shaping techniques is then demonstrated with this architecture on a visual tracking task. Robot controllers are evolved initially in simulation and then are transferred to a real robot. Performance comparisons are made with a hand-crafted controller for the same task.

1 Introduction

For many years now, automatic design approaches based on techniques from Genetic Algorithms, Reinforcement Learning and Neural Networks, have been touted as the way in which the engineers of the future will produce robot control systems. Just specify at a high level what you want the robot to do, and then sit back and wait while the GA/RL/NN works out how to get the robot to do it. While this is undoubtedly a very attractive goal, a quick look at the current state of the art of automated robot design reveals that we are a long way from being able to automatically design controllers that can out-perform human-designed ones.[1] While robot controllers have been learned from scratch for very simple 'insect-like' behaviours, general-purpose *tabula rasa* learning of complex tasks seems difficult or impossible.

One solution that has been proposed many times is to bring the human back into the design cycle: rather than attempt to learn a complex task in one go, learn it in an incremental, hierarchical fashion, with the human providing a task-specific framework, and the learning system 'filling in the blanks'. Examples include (Connell and Mahadevan, 1993; Dorigo and Colombetti, 1993; Lee et al., 1997). Typically these approaches require a human to decide in advance how to decompose the controller into sub-modules.

A criticism that is often levelled at these approaches is that they require the human designer to engage in exactly the kind of low-level analysis that we're supposed to be avoiding by using automated design in the first place! The argument goes that if the human is able to specify the detailed function of all the modules and their interconnections, internal representations, etc., then in fact they probably have a very good idea exactly how the control-

[1] A strong statement, so I'll qualify it with: 'except in a few very well specified domains where the structure of the controller is well known in advance ...'

ler is going to work. Hence they could better spend their time simply directly implementing that solution on the robot, rather than laboriously and carefully fine-tuning the learning components to produce the result they've already decided upon. Additionally, if this criticism is valid, then it suggests that we will be unable to ever evolve robot controllers significantly more proficient than ones we can design.

Our feeling is that some of these criticisms could be answered by using a learning architecture that explicitly imposes a barrier between the human designer and the internal workings of the controller. Such a system would allow the human to influence the behaviour of the controller in external/high-level ways, but would disallow tampering with internal connections, representations and so on. Although it seems crazy at first to hamper the designer like this, the hope is that by forcing us from the start to develop systems that can be trained in a high-level way, we may end up with a learning methodology which is much more scalable than existing approaches.

If we are to achieve this goal, then we need an architecture that can not only learn individual behaviours, but that can also figure out by itself how to connect those behaviours up in a suitable hierarchy and that can decide what, if any internal representations to use for messages between individual behaviours.

To make things tractable, this architecture should be constructed in an incremental fashion. The designer must devise a training programme that starts off with the robot learning to perform simple skills in simple environments, and that gradually increases in complexity up to the required task. In response to this incremental training programme, we wish the learning architecture to incrementally build a suitable modular controller that meets the requirements of the training programme at each step along the way.

Note that the designer is only responsible for decomposing the training programme into simpler units — it is the job of the learning architecture to decide how to reflect this in terms of modular units in the controller. Ideally, the designer does not interfere directly with the internal workings of the controller at all.

The approach is somewhat similar to training an animal to perform a complex task. An animal trainer has no direct access to the internal workings of the animal's brain, so for tasks that are too complex to learn in one go, the trainer must design an incremental training programme that starts with the animal learning simple tasks and proceeds to more complex tasks that build on those simple tasks. In animal psychology, such incremental training is called 'shaping' (Skinner, 1938). This term has been used in robotics by a number of people, in particular by Marco Dorigo (Dorigo and Colombetti, 1993). Our use of the word differs very slightly from his in that we place emphasis on treating the learning controller as a true 'black box', as is the case when shaping animal behaviour. See (Perkins and Hayes, 1996; Perkins and Hayes, 1997) for a more detailed discussion of the philosophy behind our approach to shaping.

Training a robot using shaping requires a learning architecture that is capable of spontaneously forming a modular, hierarchical (or even heterarchical) controller in response to a progressive training programme. The controller must not forget skills it has already learned, even if they are not directly relevant to the current phase of the training process, and it must be able to opportunistically combine previously learned skills together to form new more complex skills. The remainder of this paper sets out the architecture which we have devised for this purpose, and reports the results of experiments in simulated and real environments.

2 Task Domain

2.1 Visual Tracking

If we are going to demonstrate the usefulness of shaping, then we need a reasonably complex task domain. Tasks involving vision are notoriously hard for learning systems due to the vast quantity of data that video cameras produce each second, coupled with the relatively high difficulty in interpreting visual pixel data, compared with, for instance, sonar echoes.

Tracking moving objects using vision is a basic skill in most visually equipped animals and has been widely studied in the robotic vision community. A robotic 'camera head' that can fixate on moving objects is a commonplace sight in robotics labs around the world, but virtually all of these have been carefully programmed by hand. Our ultimate goal then is to get a robot to learn a controller that can reliably perform this task, with little or no preprocessing of the visual data.

At the present time however, we are concentrating on the substantially easier task of tracking a bright light against a moderately dark background. Such a behaviour can in fact be programmed into a robot relatively easily, but the task is useful to test out the basic ideas.

Our robotic platform consists of an RWI B21 mobile robot equipped with a single colour video camera mounted on a Directed Perception pan/tilt unit (PTU). The behaviour we require is that the robot should keep the brightest object in front of it centred in its visual field.

2.2 A Minimal Visual Simulator

Learning algorithms typically take a long time to run. This problem is often magnified for robots since the speed at which controllers can be evaluated is limited by the time it takes the robot to physically move. Therefore in common with many other researchers, e.g. (Lund and Miglino, 1996; Jacobi, 1997), we have decided to train the controller in simulation initially and then attempt to transfer it directly to the real robot (perhaps followed by a small amount of additional training). Using simulations has an additional advantage: it allows us to use more 'informed' evaluation functions that can make learning much faster.

Simulations are often criticized on the grounds that it is very unlikely that a controller evolved to work in a simulator will ever work anywhere else due to the impossibility of accurately modelling the real world. However, several researchers (Jacobi, 1997; Miglino et al., 1995) have shown that with a little care, it is possible to perform this simulation to reality transfer.

At first thought, it might seem that the 'best' simulator possible is the one that most closely models the real world. Perversely however, it turns out that in fact you can do better by making the simulation worse. Jacobi calls this the 'radical envelope of noise hypothesis' and it has three main tenets: (1) For features of the real world that are relevant to the behaviour, model them as closely as possible and then add a bit of noise to account for the modelling inaccuracy; (2) For features in the real world that are not relevant to the behaviour, model them poorly and add lots of noise to prevent the learning system using those features; (3) Don't bother to model eventualities which are not part of the target behaviour since they won't be encountered when that behaviour has been learned correctly.

One problem with this approach is that it requires the designer to decide in advance which parts of the environment to model accurately, and which to leave out, and it is often difficult to do this without having at least some idea how the robot is going to solve the problem being set. So this strategy does put some additional demands on the designer, but the hope is still that for complicated tasks, the advantages in using simulations outweigh the additional design complexity.

Following from this, our simulator doesn't attempt to model a realistic scene — instead, the image produced by the simulator consists of a completely random background, with a fairly crudely modelled bright spot somewhere in it. Figure 1 shows a robot's-eye view.

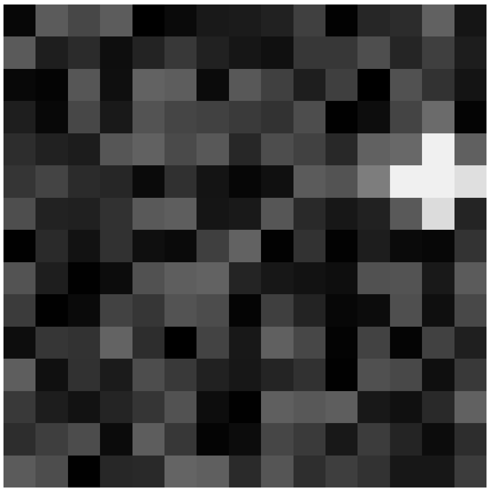

Figure 1: 15×15 resolution image of a typical 'minimally simulated' scene.

3 Architectures

In this section we will introduce the basic monolithic learning system that is at the heart of our shaping system. Successive sections will expand this architecture into something we can do real shaping with.

3.1 What Learning Algorithm?

There are a number of learning systems which could be used in shaping framework, but we have decided to opt for a genetic algorithm. There are several reasons for this.

Firstly, complex robotic tasks often fall into a 'reinforcement learning' framework, where the only feedback available is some measure (typically scalar) of how well the robot is doing, rather than a 'supervised learning' framework, where the trainer knows exactly what the robot should be doing in a number of example cases. This largely precludes techniques such as classic neural network gradient descent.[2]

Secondly, we decided from the start that we wanted to deal with continuous state and action spaces. This makes using classic 'reinforcement learning' techniques involving value functions, such as Q-learning (Watkins, 1989), problematic, although there are adaptations of the basic methods that can work in such continuous spaces (Gullapalli, 1990; Santamaria et al., 1996).

Genetic algorithms however are well suited to working in continuous spaces and with scalar evaluation. There is some scope for adding in ideas from RL however, and this will be discussed in Section 9.

3.2 What Representation?

There is then the question of what to evolve with the GA. Many researchers have used GAs to evolve neural network controllers, e.g. (Nolfi and Parisi, 1995; Floreano and Mondada, 1994). NNs have the advantage that they are relatively homogeneous and hence easy to encode into a GA. However, we prefer to use a genetic programming approach (Koza, 1992). Partly this is because we find that the structures produced by GP are easier to analyse than typical evolved NNs. In addition, GP allows us to easily add arbitrary function units into the mix if we have good reason to believe that those functions might be useful in the task being learned (the GA is under no obligation to use them of course). Finally we have a sneaking suspicion that the popularity of NNs in GAs largely derives from their use in supervised learning systems, where their simplicity and homogeneity is essential for gradient descent to be feasible. GAs do not require these properties and our feeling is that GP provides a richer, more expressive architecture, in which solutions can be expressed with fewer parameters than with NNs.[3]

Traditional GP evolves function trees. The terminal nodes of the tree represent inputs to the system (sensor values in the case of robots) or constants, and the tree produces a single output. The major evolutionary operator is 'sub-tree crossover' where randomly selected sub-trees from two parents are swapped to produce two new children.

One problem with canonical GP is that a single function node can only provide its output to one other function node. In practice if we've gone to the trouble of computing some value, we might well want to use that same value elsewhere in the tree. Canonical GP requires us to recalculate that value each time it is used. Another problem is that we would often like to get more than one output from an evolved controller. We could evolve several separate trees, but this is wasteful if the trees end up separately learning how to compute a quantity that in fact all the trees require.

For these reasons we use a variant of GP that allows arbitrary acyclic function graphs to be represented and provides an easy way of getting more than one output from the graph. In our system, illustrated in Figure 2, programs consist of ordered linear strings of function nodes, with each node taking its inputs from nodes to the right of it. Input connections can skip arbitrary numbers of nodes, as long as they connect to nodes to the right of their owner. Nodes can be connected to by an arbitrary number of other nodes. For each output required from the graph, a node is chosen to represent that output. As in a normal GP function tree, the value of a node is calculated in a recursive fashion by first calculating the values of any nodes it takes input from and then performing whatever function the node represents on those values. This recursion bottoms out when a terminal node such as a sensor or constant is encountered and so to ensure the function graph is well-formed, the rightmost node must be a terminal node. Nodes cache their values to avoid unnecessary duplicate computation.

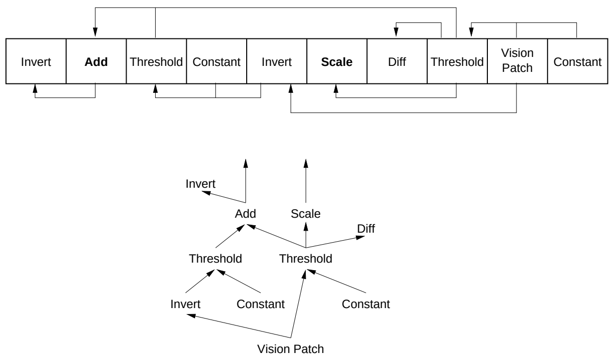

Figure 2: Basic architecture. The figure shows the linear genome representation used in our GP at the top and the equivalent function graph underneath. Bold lettering denotes nodes selected as output nodes.

Our representation means that it is common that there are redundant nodes in the function graph

[2] Although see Complementary Reinforcement Back Prop (Ackley and Littman, 1990) for something similar to gradient descent using reinforcement signals and binary neural networks.

[3] Particularly where the structure of the network must be evolved as well as the neural weights.

that are not involved in calculating the output. We do not see this as a disadvantage however, since this 'junk DNA' provides a way in which genetic diversity may be preserved in the genome - an important factor in GAs.

The tracking task is a visual processing problem. Architectures for performing visual processing typically involve carrying out the same process on many different parts of an image, or equivalently, having many identical processes each acting on different parts of the image. It would be nice if our GP system provided a way of efficiently encoding this kind of replication. We can achieve this by introducing a 'visual receptive field' terminal node, which, instead of returning just the value of a single pixel in the image, returns an array of values corresponding to some region of the image. If we then adapt our functions so that they can operate on arrays as well as scalars then our simple function graphs can represent functions with a high degree of replication.

A final difference between our architecture and canonical GP is that we rely much more heavily on mutation than on crossover. There is considerable debate in the GP community as to whether crossover is anything more than a heavy mutation operator. Other researchers have also decided to manage without crossover, e.g. (Angeline, 1997), and our experiences tend to confirm theirs.

4 Experimental Setup

4.1 Sensors and Actuators

A successful controller has to learn how to link sensing to action in a way that performs the desired behaviour. On the sensing side, our controllers are provided with access to the image data from the camera at a simulated 4Hz frame rate. The camera image contains $240 \times 240 = 57600$ pixels. Before being presented to the controller, pixel intensities are converted into real numbers ranging from 0.0 (black) to 1.0 (white). The `Vision` sensor function allows controllers to access this raw pixel data at a range of resolutions. Low resolution pixels are produced by averaging blocks of raw image pixels.

On the action side of the controller sits the PTU, which in our setup is 'speed-controlled', with independent control of pan and tilt axes. The controller outputs two real numbers between 0.0 and 1.0, one for the pan axis, one for the tilt axis. 0.0 is interpreted at maximum speed left/down, 1.0 is interpreted as maximum speed right/up and 0.5 is interpreted as stationary.

4.2 Evolutionary Details

For reasons that will become clear later, we use the term 'agent' to refer to an individual controller in the GA population.

Table 1 shows the function set that we used in our experiments. All sensors produce real values between 0.0 and 1.0 and all functions output real values in the same range. Many functions have internal parameters that determine their operation that are subject to mutation. Unless otherwise specified, functions operating on arrays simply output identically sized arrays where each value in the output array represents the result of applying the function to the corresponding value in the input array. If a particular input to a function is described as 'scalar', then if it is passed an array of values, those values are first averaged to produce a single number.

Function	Arity	Description
`Constant`	0	Constant value between 0.0 and 1.0
`Vision`	0	Array of values from a rectangular section of the image; locations, size and resolution described by internal parameters.
`Invert`	1	Output is (1.0 - input)
`Average`	2	Output is half the sum of the two scalar inputs
`Threshold`	2	If input 1 is less than scalar input 2, then output is 0, else output is 1.
`Scale`	2	Output is input 1, multiplied or divided by scalar input 2

Table 1: Functions used in visual tracking experiments

A key part of any evolutionary algorithm is the evaluation function. In our case agents are evaluated by starting the simulator in a semi-random position which ensures that the target is towards the edge of the visible image. The simulation is then run for 10 simulated sense/act cycles and the evaluation given is the average distance of the target from the centre of the image during that run, weighted slightly towards the later parts of the run. This measure is then scaled and negated so that the maximum possible evaluation is 1.0 if the agent keeps the target exactly centred throughout the entire run.

Since the simulation and starting states are random, there can be considerable variation in different evaluations of the same individual, and to counter this we evaluate each agent a number of times with different starting states and average the scores. This raises the question as to how many time to evaluate each agent — too few and the fitness evaluations will be essentially random and the GA may perform badly, too many, and we will be wasting valuable CPU cycles. A convenient solution is provided by Teller and Andre's RAT (Rational Allocation of Trials) mechanism (Teller and Andre, 1997), in which agents in a tournament are only evaluated until it is statistically clear that they have won, or it is clear that they will never win — remember it is only the winner of the tournament we are interested in. We use a modification of RAT adapted for steady-state GAs rather than the original generational framework. Using RAT to dynamically choose the optimal number of trials typically resulted in about 30-40 evaluations per tournament being required in our experiments.

Two variants of the basic task were used. In the first, which we call the 'reduced task', the agent only has to learn how to control the pan axis of the PTU to get maximum reward. Any tilt offset of the target is ignored in the evaluation. In the 'full task', both pan and tilt axes must be controlled correctly for maximum reward.

All our experiments used a GA population size of 100. Individuals are initially randomly generated with a random number of functions between 1 and some preset maximum: F_{max}.

A localized tournament selection method was employed. The 100 members of the population are arrayed on a 10×10 grid with toroidal wraparound. A single tournament consists of selecting a random starting individual, and then performing a random walk of fixed size. All the individuals encountered along the walk are evaluated and the best one is picked out. Crossover between this best individual and another individual randomly chosen from the rest of the tournament is carried out with (low) probability P_c. The resulting individual is then mutated and replaces the randomly chosen individual. For all of our experiments a tournament size of 4 was used.

Table 2 summarizes the different genetic operators we used and their default probabilities.

Except where noted, experiments ran for 15,000 tournaments.

4.3 Experimental Results

Figure 3 shows the results of the initial experiments on the reduced 'pan-only' task. The graph plots the fitness of the fittest individual in the population against generation. As with all the experiments

Operator	**Probability**
Crossover (P_c)	0.02
Mutate a function internal parameter	0.5
Randomize a function's internal parameters	0.02
Delete/insert a function	0.02
Completely randomize	0.02

Table 2: Mutations used in visual tracking experiments. Note that if an agent contains no functions with mutatable parameters, then a randomly chosen function is randomized or inserted/deleted instead.

reported in this paper, the plot indicates the distribution of maximum fitnesses from 50 different evolutionary runs using different random number seeds. The central solid line shows the median of this distribution, while the dotted lines above and below this line show the 90th and 10th percentiles of the distribution. For the experiment shown, $P_c = 0.02$ and $F_{max} = 20$.

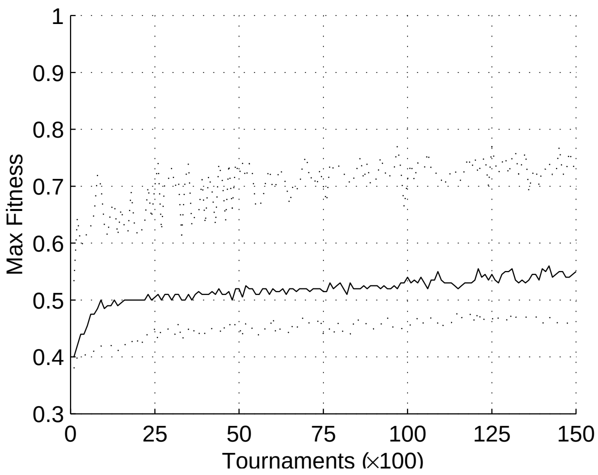

Figure 3: Fitness vs. generation for the basic architecture on the reduced task with 20 functions per agent.

The performance of the final evolved agents was not terribly impressive. We experimented with different values of P_c between 0.02 and 0.5, but found that the only significant variation was that larger values gave slightly poorer results. We also explored a variety of values for F_{max} between 5 and 40 and again found no significant variation in the results, with 20 functions per agent perhaps performing slightly better than the others.

5 Rule-Based Genetic Programming

5.1 Architecture

Why does the basic architecture fail to find an acceptable solution and how can it be improved? Perhaps the problem is just too difficult for a single monolithic agent. Our shaping approach is based on ideas of 'divide-and-conquer' and specialization, so why not apply these ideas to the basic architecture as well?

Figure 4 shows an enhancement of the basic architecture that we call 'Rule-Based Genetic Programming'. Instead of attempting to get a single function tree to learn the entire target behaviour, we split the agent into a number of distinct function trees called *rules*. Each rule looks exactly like the function trees described earlier and produces two outputs. The first of these is called the 'validity' and is intended to express how appropriate that rule is to the current situation. The second output specifies the value that would be sent to the rule's actuator if that rule was chosen. Rules are randomly assigned to different actuators - in our case to control either the pan axis of the PTU, or the tilt axis. At each cycle, all those rules in the current agent that are competing to control a particular actuator are compared, and the one with the highest validity is chosen. The value of that rule is then sent to its actuator. If no rules are competing to control a particular actuator then that actuator carries out a default action, which in our case corresponds to a value of 0.5 (zero pan or tilt speed).

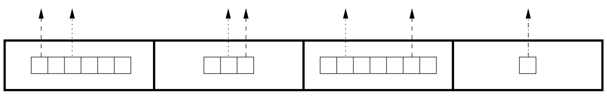

Figure 4: Rule-Based Genetic Programming. Shown here is an agent with four rules, three assigned to control the pan axis, and one to the tilt axis. Each rule is a function graph with two outputs: a validity, indicated by a dotted arrow, and a value, indicated by a dashed arrow.

The operation of this system is somewhat like a Pittsburgh-style classifier system (Smith, 1983), except that the condition part of each rule is computed using a function graph (the validity), rather than by matching a bit-pattern as in traditional classifier systems, and the action is a computed value, rather than a fixed one. We also do not make use of a rule strength — instead, the validity is used to decide which rule to use — but see Section 9 for a brief discussion of experiments where we added Q-learning to these rules.

There is another good reason for using rules. In traditional GAs, the crossover operator is widely used and appears to do a useful job. However, crossover is less convincing when used with encodings where epistasis is very high, such as those used in GP, including our basic architecture. This has lead several researchers to abandon crossover in favour of various forms of mutation when using these encodings, e.g. (Jones, 1995; Angeline, 1997). Interestingly, about 10 years ago, Kenneth de Jong wrote about the choice of language for an evolvable programming system (De Jong, 1988):

> "...there is at least one important feature that Lisp shares with other more traditional languages: they are all procedural in nature. As a consequence most reasonable representations have the kinds of properties that cause considerable difficulty in GA applications."

He then goes on to suggest that production systems are surely the correct representation to use with GAs. In our architecture we re-introduce crossover in a place where it will be useful (at the rule level) rather than damaging (at the function graph level). Hence we can hope to get the best of both worlds — the expressiveness of GP combined with the evolvability of production systems.

In the new system, agents are evaluated as before and two parents are selected as described for function graph crossover. Then we form a single child from the parents using uniform crossover — i.e. for each particular rule slot there is an equal chance of that rule coming from each parent. All agents in a population have the same number of rules. Uniform crossover is not widely used in most GA work, since it is very likely to scramble schemata of any length. Here however, our fundamental schemata are the function graphs that make up individual rules, and rule-level crossover never breaks those up.

5.2 Experimental Results

Figure 5 shows the results of the rule-based architecture on the reduced task, with 4 rules per agent. For each rule, F_{max} was set to 5 and the GA was run for 150 generations. No significant difference was observed as the number of rules per agent was varied from 3 to 5. The results are much better than with a single rule per agent: the mean max fitness with single rule agents is 0.57, while with 4 rules per agent it is 0.81. The two architectures were compared using a one-tailed non-parametric randomization test (Cohen, 1995) and the second system was found to be significantly better ($p < 0.01$).

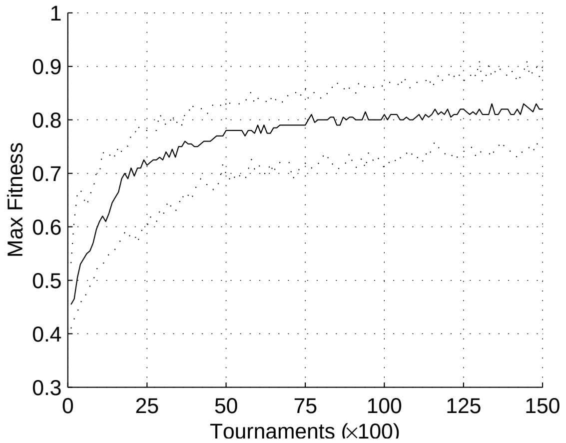

Figure 5: Fitness vs. generation for the rule-based architecture on the reduced task with 4 rules per agent.

Figure 6 shows the results of the same architecture on the full task. Since this problem was harder, we ran the GA for 300 generations and tried using both 4 and 8 rules per agent. Here the results are less convincing — the results are similar no matter how many rules were used (the graph shows the results for 8 rules per agent) and the best individuals from the best runs had fitnesses of only just over 0.7.

6 Shaping

6.1 Architecture

The architecture described so far represents the core of our shaping system. Recall that shaping involves a human designer setting up a training programme which involves learning a series of progressively more difficult tasks until the target task is acquired. The basic learning system must be capable of learning the small increments by itself (perhaps making use of previously learned skills), and so the power of this learning system determines how fine-grained the training programme must be.

The system described above can evolve an agent that can satisfy the first increment on a training programme. To extend it to something that can

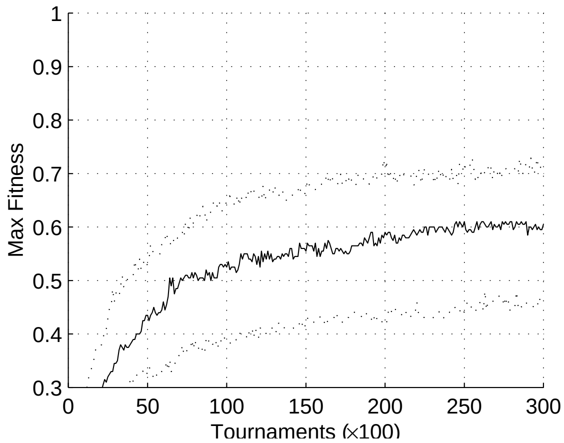

Figure 6: Fitness vs. generation for the rule-based architecture on the full task with 8 rules per agent.

learn multiple increments we extend the system to allow multiple agents. At the time of writing, multiple agents within a controller act independently, with more recently evolved agents overriding other agents in case of conflict. In the future we intend to add a mechanism whereby agents can excite or inhibit each other, but this is not necessary for the current task.

According to our shaping philosophy, the actions of the designer are limited to (a) adjusting the evaluation criteria and the external environment, and (b) telling the controller that it should prepare itself to learn a new skill. The designer is not allowed to specify internal representations or specify how agents connect to each other or to sensors and actuators. In this way we are treating the learning system as a black box. When the system is told to prepare for a new learning phase, it 'freezes' the agent most recently learned, and adds a new agent to the controller which will be evolved by the GA to work with the existing agents in whatever way is appropriate.

6.2 Experimental Results

To demonstrate the utility of shaping we took the 90th percentile 'best' controller[4] from the previous experiments on the reduced task with 4 rules per agent, froze it, added a second unevolved agent, and then ran the GA for 150 generations using the

[4]We use the 90th percentile best controller rather than the absolute best one, because we can be confident that controllers of at least this fitness are easily and reliably obtained by doing a relatively small number of runs, while our absolute best one might be a rare fluke.

full task evaluation function to evolve that second agent. Note that the GA had already been run for 150 generations to generate the first agent.

Figure 7 shows the maximum fitness distribution over 50 runs.

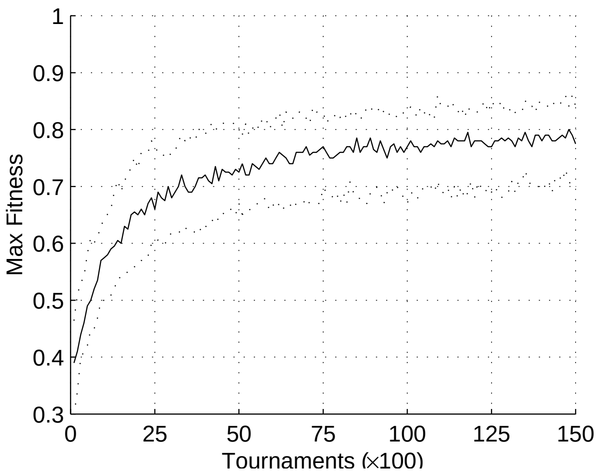

Figure 7: Fitness vs. generation for the multi-agent shaped architecture on the full task.

Compare this graph with the single agent runs on the full task using 8 agents shown in Figure 6. Remember that the latter runs are 300 generations long, whereas the 'shaped' graphs are for 150 generations after an initial 150 generations.

The 'shaped' two-agent controller achieved a significantly better final performance than the 'non-shaped' single-agent controller: a mean max fitness of 0.76 rather than a mean max fitness of 0.6, with $p < 0.01$ that the first one is higher using a one-tailed randomization test.

7 Expert Controllers and Real Robots

The fitness values quoted so far allow us to compare the relative performance of various forms of the GA in simulation, but two questions remain unanswered: how does the performance of the evolved controller compare with a hand-crafted controller?, and: does the evolved controller work on a real robot?

To answer the first of these questions we hand-crafted a controller to solve the same light tracking task that was put to the GA. For this simple problem we readily admit that designing a controller by hand is substantially easier than evolving one! The promise of shaping lies in incrementally evolving complex controllers that we can not easily design by hand, but the current task is still useful as an illustration.

We tested the hand-crafted 'expert' over 100 trial runs of the simulator and observed a mean performance of 0.91. This is not much higher than the 90th percentile best evolved controller seen before, with a fitness of 0.85, and shows that in the simulated environment at least, our evolved controllers can compete with hand-crafted controllers.

We then carried out a comparison between the 90th percentile best evolved controller and the hand-crafted expert on the real robot. A stationary light was successively moved through 50 different positions in front of the robot, and at each position the controller was evaluated on its ability to move the camera from a 'home position' to point at the light. Since the light positions were difficult to measure accurately, we used the expert's ability to locate the light source to provide an evaluation function. Obviously this means that the expert is going to score rather well since it is effectively evaluating itself! In fact, the evolved controller had a mean performance of 0.61, while the expert had an mean performance of 0.84.

The performance of the evolved controllers is perhaps a little disappointing, especially given the simplicity of the hand-crafted controller! It is interesting to look at the way in which the evolved controllers failed though: for most test problems, the evolved controllers performed as well as or better than the hand-crafted expert. However, for several clusters of test problems, they performed extremely badly, giving the performance distribution a very long left-hand tail, which drags down the mean performance significantly. The median performance of the evolved and expert controllers was closer (0.75 and 0.86 respectively) since the median is less affected by the few low outliers. What this suggests is that we need a way to ensure that the evolved controllers cannot 'get away' with just performing well on a few test cases. This could perhaps be done by dynamically selecting fitness cases on the basis of how badly controllers have performed on them in the past, e.g. as in (Gathercole and Ross, 1994).

8 Related Work

As discussed at the start of paper, several other researchers use the term 'shaping', principally Marco Dorigo (Dorigo and Colombetti, 1993; Dorigo and Colombetti, 1998). There are many similarities between our work and his — a multi-agent architecture, some sort of staged learning and an evolutionary learning system — and our work can be seen as an extension to reduce further the required human design input.

Ulrich Nehmzow (Nehmzow and McGonigle, 1994) uses the word 'shaping' somewhat differently to refer to external reinforcement provided by a human rather than by some automatic reinforcement program.

Training decomposition is only one way of 'shaping' behaviour without delving into the learning system's innards. Another important method is to influence the environment to make carrying out a task easier or harder. By starting on easy problems and moving to harder problems, it may well be possible to learn faster than if the robot was presented with only the hard problems right from the start. Asada calls this 'learning from easy missions' (Asada et al., 1996).

Several other researchers have looked at using multiple GP trees in concert to improve performance on complicated tasks, e.g. (Luke and Spector, 1996; Haynes and Sen, 1997), but not in a shaping context as far as we know.

There has also been some work on getting GP to learn visual processing (Poli, 1996; Tackett, 1993), but we are not aware of any work using it for visual tracking tasks.

Our rule-based system has many things in common with classifier systems, about which there is a vast literature dating back to their beginning with John Holland (Holland and Reitman, 1978). Inspirational works include research by Potter (Potter et al., 1995), Wilson (Wilson and Goldberg, 1989; Wilson, 1995) and Bonarini (Bonarini, 1995).

There has been at least one example of evolving a neural network controller to do visual tracking of moving objects in experiments performed by Nick Jacobi, but as far as we know there is nothing in print about this yet. Jacobi pre-processes the image data from the camera by performing frame differencing on it initially, which means that moving objects appear as bright patches in the image presented to the neural network. Hence this controller seems to be doing a form of light tracking. The controller learns to generate quick 'saccades' which centralize any disturbances in the visual field. As far as we can see this technique could not be used for smooth tracking.

9 Conclusions and Future Work

The experiments reported here have shown that it is possible to learn a simple visual tracking task moderately well using a rule-based GP-like architecture, and that the rule-based architecture is significantly better than using a monolithic GP architecture. We have also shown that the use of a staged learning process can lead to better controllers without the designer having to tinker with the controller internals. However there is still much to do.

There are a number of extensions to the basic architecture that we are considering. One that has already been hinted at is to introduce some ideas from the fields of Reinforcement Learning and Classifier Systems. Essentially this would be used within an agent to learn when to fire a particular rule and would replace the current evolved validity mechanism. One such scheme first uses the existing validity to decide whether a rule is active or inactive (represented by whether the validity is above or below 0.5). In this way the rule is associated with a genetically determined sub-region of the state space. The Q-value of a rule is then defined as the expected total discounted reward that would be obtained by following the rule's action (i.e. using its value to control an actuator) when we are in its part of the state space. If we could learn this value then we could easily choose rules by simply picking the one with the highest Q-value out of the currently active set. Unfortunately the experiments to date have been at best inconclusive and more often have shown that this extension makes things worse! It is well-known that Q-learning is not guaranteed to converge when function approximators such as our rules are used and in addition it appears that the learning mechanism interferes with the evolutionary mechanism since the timescales on which they work are too similar. More work needs to be done to clarify the situation.

More work need to be done to ensure that the GA evolves controllers that work robustly in all commonly encountered situations. We are currently exploring methods involving dynamic fitness case selection.

Finally as we mentioned at the start of the paper, our eventual aim is to have a robot that can fixate on moving objects. This will involve a much more complicated controller with significant inter-agent interaction.

References

Ackley, D. and Littman, M. (1990). Generalization and scaling in reinforcement learning. In Touretsky, D., editor, *Advances in Neural Information Processing Systems 2*. Morgan Kaufmann, San Mateo, CA.

Angeline, P. (1997). Subtree crossover: Building block engine or macromutation. In *Genetic Programming 1997*, Stanford University. Morgan Kaufmann.

Asada, M., Noda, S., Tawaratsumida, S., and Hosoda, K. (1996). Purposive behaviour acquisition for a real robot by vision-based reinforcement learning. *Machine Learning*, 23.

Bonarini, A. (1995). Evolutionary learning of fuzzy rules: Competition and cooperation. In Pedrycz, W., editor, *Fuzzy Modelling: Paradigms and Practice*. Kluwer Academic Press.

Cohen, P. (1995). *Empirical Methods for Artificial Intelligence*. MIT Press.

Connell, J. H. and Mahadevan, S. (1993). Rapid task learning for real robots. In Connell, J. H. and Mahadevan, S., editors, *Robot Learning*, chapter 5, pages 105–139. Kluwer Academic Press.

De Jong, K. (1988). On using genetic algorithms to search program spaces. In *Proc. 2nd Int. Conf. on Genetic Algorithms*.

Dorigo, M. and Colombetti, M. (1993). Robot shaping: Developing situated agents through learning. Technical Report TR-92-040, International Computer Science Institute, Berkley, CA 94704.

Dorigo, M. and Colombetti, M. (1998). *Robot Shaping: An Experiment in Behaviour Engineering*. MIT Press/Bradford Books.

Floreano, D. and Mondada, F. (1994). Automatic creation of an autonomous agent: Genetic evolution of a neural network driven robot. In D. Cliff, P. Husbands, J.-A. M. and Wilson, S., editors, *From Animals to Animats 3: Proc. 3rd Int. Conf. Simulation of Adaptive Behavior*, Cambridge MA. MIT Press.

Gathercole, C. and Ross, P. (1994). Dynamic training subset selection for supervised learning in genetic programming. In davidor, Y., Schwefl, H.-P., and Männer, R., editors, *Parallel Problem Solving from Nature III*, pages 312–321. Springer-Verlag.

Gullapalli, V. (1990). Associative reinforcement learning of real-valued functions. Technical Report 90-129, COINS, Univ. of Massachusetts, Amherst, MA.

Haynes, T. and Sen, S. (1997). Crossover operators for evolving a team. In *Genetic programming '97*. Morgan Kaufmann.

Holland, J. and Reitman, J. (1978). Cognitive systems based on adaptive algorithms. In Waterman, D. and Hayes-Roth, F., editors, *Pattern Directed Inference Systems*, pages 313–329. Academic Press, New York.

Jacobi, N. (1997). Evolutionary robotics and the radical envelope of noise hypothesis. *Adaptive Behavior*, 6.

Jones, T. (1995). Crossover, macromutation and population-based search. In *Proc. 6th Int. Conf on Genetic Algorithms*. Morgan Kaufmann.

Koza, J. (1992). *Genetic programming: On the Programming of Computers by Means of Natural Selection*. MIT Press.

Lee, W.-P., Hallam, J., and henrik Lund (1997). Applying genetic programming to evolve behaviour primitives and aribtrators for mobile robots. In *Proc. 4th Int. Conf. on Evolutionary Computation*. IEEE Press.

Luke, S. and Spector, L. (1996). Evolving teamwork and coordination with genetic programming. In *Genetic Programming '96*, Stanford University.

Lund, H. and Miglino, O. (1996). From simulated to real robots. In *Proc 3rd IEEE Conf. on Evolutionary Computation*. IEEE Press.

Miglino, O., Lund, H., and Nolfi, S. (1995). Evolving mobile robots in simulated and real environments. *Artifical Life*, 2(4).

Nehmzow, U. and McGonigle, B. (1994). Achieving rapid adaptations in robots by means of external tuition. In Cliff, D., Meyer, J.-A., and Wilson, S., editors, *From Animals to Animats 3: Proc. 3rd Int. Conf. Simulation of Adaptive Behavior*. MIT Press.

Nolfi, S. and Parisi, D. (1995). Evolving non-trivial behaviours on real robots: An autonomous robot that picks up objects. In Gori, M. and Soda, G., editors, *Proceedings of the fourth congress of the Italian Association of Artificial Intelligence*, pages 243–254. Springer-Verlag.

Perkins, S. and Hayes, G. (1996). Robot shaping — principles, methods and architectures. Technical Report 795, Dept. of Artificial Intelligence, Univ. of Edinburgh, Scotland.

Perkins, S. and Hayes, G. (1997). Incremental acquisition of complex behaviour using structured evolution. In *Proc. Int. Conf on Genetic Algorithms and Neural Networks*. Also available from: http://www.dai.ed.ac.uk/students/simonpe/.

Poli, R. (1996). Genetic programming for image analysis. In *Genetic Programming '96*. Morgan Kaufmann.

Potter, M., De Jong, K., and Grefenstette, J. (1995). A coevolutionary approach to learning in sequential decision rules. In *Proc. 6th Int. Conf. on Genetic Algorithms*, Pitsburgh. Morgan Kaufmann.

Santamaria, J. C., Sutton, R., and Ram, A. (1996). Experiments with reinforcement learning in problems with continuous state and action spaces. Technical Report 96-088, COINS, Univ. of Massachusetts, Amherst, MA.

Skinner, B. (1938). *The Behaviour of Organisms: An Experimental Analysis*. D. Appleton Century.

Smith, S. (1983). Flexible learning of problem solving heuristics through adaptive search. In *Proc 8th Int. Joint Conf. on Artificial Intelligence*.

Tackett, W. (1993). Genetic programming for feature discovery and image discrimination. In *Proc. 5th Int. Conf. on Genetic Algorithms*.

Teller, A. and Andre, D. (1997). Automatically choosing the number of fitness cases: The rational allocation of trials. In et al., J. K., editor, *Proc. Genetic Programming'97*. Morgan Kaufmann.

Watkins, C. (1989). *Learning with Delayed Rewards*. PhD thesis, Psychology Department, University of Cambridge, England.

Wilson, S. (1995). Classifier fitness based on accuracy. *Evolutionary Computation*, 3(2).

Wilson, S. and Goldberg, D. (1989). A critical review of classifier systems. In *Proc. 3rd Int. Conf. on Genetic Algorithms*. Morgan Kaufmann.