How Not to Be a Black-Box: Evolution and Genetic-Engineering of High-Level Behaviours*

Ik Soo Lim
LIG, DI
Swiss Federal Institute of Technology (EPFL)
CH-1015 Lausanne, Switzerland
iksoolim@email.com

Daniel Thalmann
LIG, DI
Swiss Federal Institute of Technology (EPFL)
CH-1015 Lausanne, Switzerland
thalmann@lig.di.epfl.ch

Abstract

In spite of many success stories in various domains, Genetic Algorithm and Genetic Programming still suffer from some significant pitfalls. Those evolved programs often lack of some important properties such as robustness, comprehensibility, transparency, modifiability and usability of domain knowledge easily available. We attempt to resolve these problems, at least in evolving high-level behaviours, by adopting a technique of *conditions-and-behaviours* originally used for minimizing the learning space in reinforcement learning. We experimentally validate the approach on a foraging task.

1 Introduction

Genetic Algorithm (GA) and Genetic Programming (GP) have a large number of successful applications in many domains. Much of their popularity is due to the fact that GA and GP are competitive if the space to be searched is large, is known not to be perfectly smooth and unimodal, or is not well understood, or if the fitness function is noisy, and if quickly finding a sufficiently good solution is enough[4], [6]. However, those evolved programs by GA/GP often lack of some important properties such as robustness, comprehensibility, full specification, modifiability and taking advantage of domain knowledge easily available, each of which is important for GA/GP paradigm to be more widely accepted and used.

Evolved motion-controller programs are, for example, often "brittle" in that they only work for a particular task rather than a general skill: slight perturbations may make the programs fail [2], [9]. Even though the size of them may be small, these programs are usually 'opaque' to human interpretation [2], [9], [12]. This opaqueness leads to more drawbacks. It may be difficult to tell how it will work for a given condition before actually running the program on it. It is not easy to modify an evolved program for a relevant but different task nor to encode and take advantage of some domain knowledge easily available.

The motivation behind this work, in particular, comes from the fact that a few number of GA/GP applications were done [2], [7], [12] in computer animation which is our main interest[13], but still suffer from many of these pitfalls and are not much followed-up. We attempt to resolve these problems, at least in evolving high-level behaviours, by adopting a technique of *conditions-and-behaviours* originally used for minimizing state space in reinforcement learning[5]. We use a foraging task as a test-bed for the approach.

This paper is organized as follows. In Section **Behaviours and Conditions**, we introduce the benefits of them. Section **The Foraging Problem** describes the task for which a policy has to evolve and an experimental framework for it. Section **Experiments** gives the experimental results and **Discussions** and **Conclusions** follow it.

2 Behaviours and Conditions

2.1 Why Behaviours and Conditions?

Behaviours are goal-driven control laws that achieve and/or maintain particular goals such as *homing* and *wall-following*. Abstracting away the low level details, behaviours can be used as the basic representation level for control in mobile robots [5]and computer animation [8].

Behaviours are triggered by *conditions*, predicates on sensor readings that map into a proper subset of the state space. Each condition is defined as the part of the state that is necessary and sufficient for activating a particular behaviour. The truth value of a condition determines when a behaviour

*In the *Proceedings of Genetic and Evolutionary Computation Conference (GECCO-99).*

can be executed and when it should be terminated, thus providing a set of events for the agent's control algorithm. This condition set is typically much smaller than the agent's complete state space[5].

Reformulating states and actions into conditions and behaviours effectively reduces the search space, which is then defined at a higher level of description due to the abstracting-away of the details: what GA/GP has to do is then to find a mapping from the power set of conditions to behaviours into the most effective policy for a given task (**Table1**).

2.2 More than Reducing Search Space

Since a policy defined at a high-level description is evolved rather than a mechanical motion controller of low-level states and actions, it is robust: evolved programs for mechanical motion controllers are often fragile that extra efforts have to be given to make them robust[2], [9].

While programs evolved by typical GA/GP are not usually easy to interpret[2], [12], this high level description provides an easy interpretation of an evolved policy. This is useful both in studying the evolved policy [1] and in modifying it. Those evolved by typical GA/GP are not prone to modifications so that the evolutionary process with a new fitness function may have to be re-run even for a small change in the programs' output[2].

Since a behaviour is mapped for each of the conditions, an evolved policy becomes transparent for every possible condition: a full specification of the policy! Most of programs evolved by typical GA/GP do not provide this sort of transparency so that they have to be actually run on it to see its performance for a given condition. It has to be noted that the easy interpretation does not necessarily imply this transparency. In [1], a parse tree was used for specifying an agent's policy which still provides easy interpretation of it. It is not, however, a mapping from the power set of conditions to behaviours since not only behaviour but also condition primitives comprise the nodes of the tree: in fact, one of the best evolved policy consisted of only behaviour nodes without a single condition node so that its performance was not transparent on any condition.

Since an agent's policy is specified in parallel rather than sequentially, it is easier to modify part of the policy with less affecting the rest of others. If it was sequential such as a parse tree which is skimmed through by a controller successively performing each primitive encountered in nodes[1], a change in a preceding node could affect the following nodes so that it is not so prone to modifications.

If there is domain knowledge easily available, it may be encoded in genotype and reduce search space further. Domain knowledge, if any, are usually taken advantage of in the form of fitness functions. This implicit use of domain knowledge still leaves a human user to set its corresponding parameters in fitness functions[1], [5]. If this domain knowledge can be expressed in terms of conditions-and-behaviours as in the following section, this can easily be encoded as part of genotype in the beginning and the search process has to fill in only the rest of others.

We experimentally validate these points on a foraging problem.

3 The Foraging Problem

Foraging problem's goal is to make an agent find and take home samples in an unpredictable environment. This biologically inspired problem serves as a canonical abstraction of a variety of real-world applications such as demining and toxic clean-up[5] and has been used in a lot of possible derived applications in artificial intelligence and artificial life (for more reference, see [1]).

The basic behaviour repertoire, given to the agent a *priori*, consists of the following fixed set:

- *homing* : move to a home base
- *grasping* : grasp a sample
- *dropping* : drop a sample
- *wandering* : move to a random location

These behaviours are 'protected' as protected division % is typically pre-set to return, say 1, when divided by 0 in GP[4]: *homing* does nothing if it is called when no home base is seen. Similarly, *grasping* and *dropping* do so for no sample seen.

A simple GA was used to search the appropriate conditions for triggering each of the above behaviours. Since only the space of conditions necessary and sufficient for triggering the behaviour set is considered, the state space is reduced to the power set of the following clustered condition predicates:

- *any-home?* : is any home base seen?
- *at-home?* : is the agent at a home base?
- *any-sample?* : is any sample not collected seen?
- *carrying-sample?* : is the agent carrying any sample?

Condition				Behaviour	Behaviour	Behaviour
at-home?	*carrying-sample?*	*any-sample?*	*any-home?*	*hand-written*	*evolved*	*engineered*
0	0	0	0	wandering		
0	0	0	1	wandering	homing	
0	0	1	0	grasping		!
0	0	1	1	grasping	homing	!
0	1	0	0	wandering		
0	1	0	1	homing	wandering	
0	1	1	0	wandering		
0	1	1	1	homing		
1	0	0	0	?		
1	0	0	1	wandering		
1	0	1	0	?		
1	0	1	1	grasping		!
1	1	0	0	?		
1	1	0	1	dropping		!
1	1	1	0	?		
1	1	1	1	dropping		!

Table 1: : The foraging polices. A *hand-written* one, an *evolved* one with a heterogeneous fitness function, and an genetic-*engineered*-&-evolved one with a monolithic fitness function. Since *any-home?* is never `FALSE` if *at-home?* `TRUE`, behaviour entries for these impossible conditions are left out and marked with '?'. Only the entries different from those of the *hand-written* one are shown in the second and the third policy. '!' is marked for the genetic-*engineered* parts of the third policy to be the same as their counterparts in the the *hand-written* one.

3.1 Genetic Code for an Agent's Foraging Policy

The agent's foraging policy is specified as a table consisting of columns for conditions and a column for their corresponding behaviours as in **Table 1**. An 1-d array corresponding to this behaviour column is used as the genotype: one of the simplest possible genotypes!

3.2 Fitness Function for the Foraging Policy

A monolithic function is typically used for a fitness function in GA/GP. Constructing such a monolithic fitness function could be a complicated task in some domains having dynamic features such as this foraging problem, since the environment may provide some immediate rewards and delayed reinforcement. To enable and accelerate the search process, both of heterogeneous rewarding functions and progress estimating functions were lately used which took advantage of implicit domain knowledge [1], [5].

In our experiments reported here, we used only the heterogeneous functions for the fitness function though including the progress estimating functions was reported to improve the performance of the policy further[5]. This is partly because of simplicity, but mainly because we would like to focus on those aspects mentioned earlier such as ease of interpretation and modification of the evolved policy rather than the performance improvement.

The following events produce immediate positive reinforcement:

- E_1: grasped-sample
- E_2: dropped-sample-at-home

The following event results in immediate negative reinforcement:

- E_3: dropped-sample-away-from-home

The events are combined into the following heterogeneous reinforcement function:

$$R(c) = \begin{cases} r_1 & if & E_1 \\ r_2 & if & E_2 \\ r_3 & if & E_3 \\ 0 & & otherwise \end{cases}$$

$$r_1, r_2 > 0, r_3 < 0$$

The fitness function is a sum of the reinforcement R received over time t:

$$\sum_t R(c,t)$$

4 Experiments

The agent has a limited visual depth of field with a view angle of 360 degree and the experiments were done with the following parameters:

Population = 300, Generation = 300, PCrossover = 0.7, PMutation = 0.1
r_1= 2, r_2 = 100, r_3= -2

4.1 The Basic Foraging Task

Due to notorious difficulties of evaluating this sort of performance, we define convergence as a particular desired policy as done in[5]. We simply compare a hand-written policy and an evolved one(**Table 1**): only those entries different from their counterparts in the first one are shown in the second. Since *any-home?* is never `FALSE` if *at-home?* `TRUE`, behaviour entries for these impossible conditions are left out and marked with '?' in **Table 1**. The size of the search space is then reduced from 4^{16} to 4^{12}.

Among twelve entries in the evolved policy, three are different from those in the hand-written one: $(12-3)/12 = 75\%$ of the correct policy (**Table 1**, **Figure** 1). If the progress estimating functions had been used in addition to the heterogeneous reward functions, the performance of the evolved policy might have been improved as reported in[5].

4.2 Variations on the Theme

4.2.1 Many Homes

The location of the home base is not known to the agent in our experiment: the agent has to look for it when homing while the world coordinate of the home base was available to the agent in the previous work[5]. This coordinate-free homing allows the evolved policy to be more genuine and flexible. We ran it in other situation where there are two home bases and the same policy still works well (**Figure 2**).

4.2.2 Carrying More than One Sample

During the evolution of the foraging policy, *carrying-sample?* was `TRUE` while it carried at least one sample. Since the evolved policy does not have any entry of *grasping* for conditions of *carrying-sample?* `TRUE`, the agent can bring home only one sample at a time even though it may encounter another in the mid of the way. By interpreting the condition more genuinely such as *carrying-sample(s-enough-not-be-able-to-take-any-more)?*, the same policy can allow the agent to bring home many samples at a time: two samples at a time were considered to be enough as if carrying one in the right hand and the other, the left hand (**Figure 3**).

4.2.3 Genetic Engineering rather than Rewarding

Due to this transparent representation of genotypes with conditions-and-behaviours, some of domain knowledge may be directly encoded in the genotype rather than be implicitly informed of by fitness functions. For the conditions satisfying both *carrying-sample?* `FALSE` and *any-sample?* `TRUE`, *grasping* needs to be encoded: this corresponds to the rewarding for E_1. For those of *at-home?* `TRUE` and *carrying-sample?* `TRUE`, *dropping*: this is equivalent to the rewarding for E_2. These genetic-engineered parts of a genotype are labeled as '!' in **Table 1**.

Then, a GA has to simply fill in the rest of the entries for a foraging policy whose size of the search space is reduced from 4^{12} to 4^7. The fitness function now becomes a monolithic function: the number of samples collected at the home base when a policy's running ends. A foraging policy evolved and was the same as the hand-written one (**Table 1**).

The following are some of possible benefits from this 'gene-manipulation' compared to the use of the heterogeneous fitness functions:

- less load in monitoring conditions and behaviours for the fitness evaluation during the evolution.
- less burden in choosing parameter values than that of the heterogeneous functions.
- both domain knowledge informed of and reduction of search space : only the knowledge informed of in the case of the heterogeneous functions.

4.2.4 Porting the Evolved Policies into 3D Computer Animation

Running a policy during the evolution was done in a rather crude simulation of sensing and behaviours and those two of the evolved policies in **Table 1** worked well in it. The first policy of 75% correctness, however, did not performed well when we ported these policies into 3D computer animation where more tight management of sensing and behaviours were employed such as a limited view angle of the agents. In the beginning of the animation, it seemed to be all right. When samples near the home bases were already collected, the agents collected other samples and started *homing*. But, the agent soon stopped *homing* before completing it and switched into *wandering* then *homing* and so on: they never reached a home base .

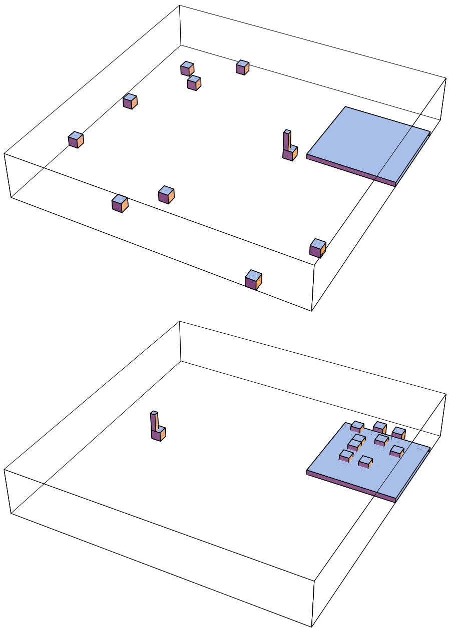

Figure 1: : The basic foraging. Clockwise, from upper left.

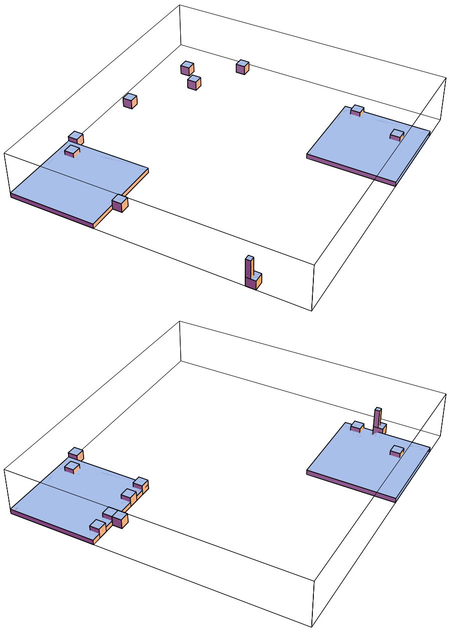

Figure 2: Many homes. Clockwise, from upper left.

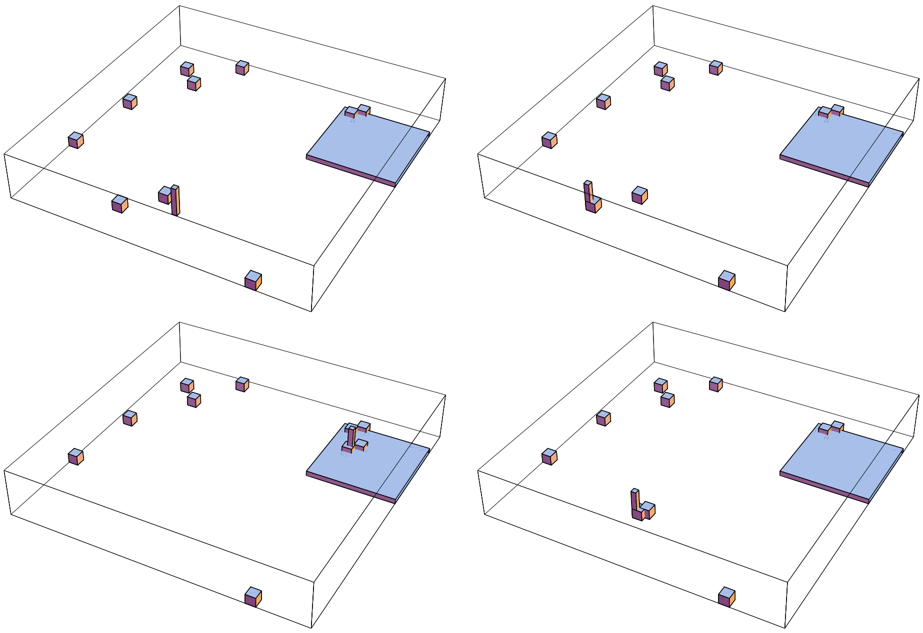

Figure 3: Carrying more than one sample at a time. Clockwise, from upper left.

We, however, were soon able to see where it went wrong and how it had to be corrected by simply monitoring the conditions and the behaviours during the animation. When the condition was 0111, it started *homing*. If no sample was seen when the agent reached near a home base, the condition became 0101 and it invoked *wandering*: the remedy was to replace it with *homing*.

The easy-to-comprehend and easy-to-modify representation of the genotype allowed us to pin down the problem of the not perfect policy and correct it. This is very important in practical use of GA/GP and black-box-like solutions evolved by typical GA/GP do not allow it.

5 Discussion and Conclusion

One problem in scaling-up the current approach is that it uses the *power set* of conditions: for n conditions, 2^n entries have to be specified. One way to overcome the problem would be to use a learning classifier system[3] where a default hierarchy is employed: a default hierarchy is a multi-level structure in which classifiers become more general as the top level is ascended. To keep the classifier system comprehensible, transparent, modifiable and usable of domain knowledge, implementing it as *stimulus-response* system[10][11] would be important.

We used conditions-and-behaviours technique to make evolved solutions by a GA robust, comprehensible, transparent, modifiable and usable of domain knowledge easily available. We experimentally validated the approach on the foraging task and the variations including its realization of 3D computer animation.

References

[1] S. Calderoni and P. Marcenac. Genetic Programming for Automatic Design of Self-Adaptive Robots. *Proceedings of EuroGP'98*, *Lecture Notes in Computer Science (LNCS 1391)*, pp. 163-177, Springer Verlag, 1998.

[2] L. Gritz and J. K. Hahn. Genetic Programming Evolution of Controllers for 3-D Character Animation. in Koza, J.R., et al. (editors), *Genetic Programming 1997: Proceedings of the 2nd Annual Conference*, pp. 139-146. July 13-16, 1997, Stanford University. San Francisco: Morgan Kaufmann.

[3] J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*, 2nd ed. Cambridge, Mass: MIT Press, 1992.

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

[5] M. Mataric. Reinforcement Learning in the Multi-Robot Domain. *Autonomous Robots*, v4 n1, pp.73-83, January 1997.

[6] M. Mitchell. *An Introduction to Genetic Algorithms.* The MIT Press, 1996.

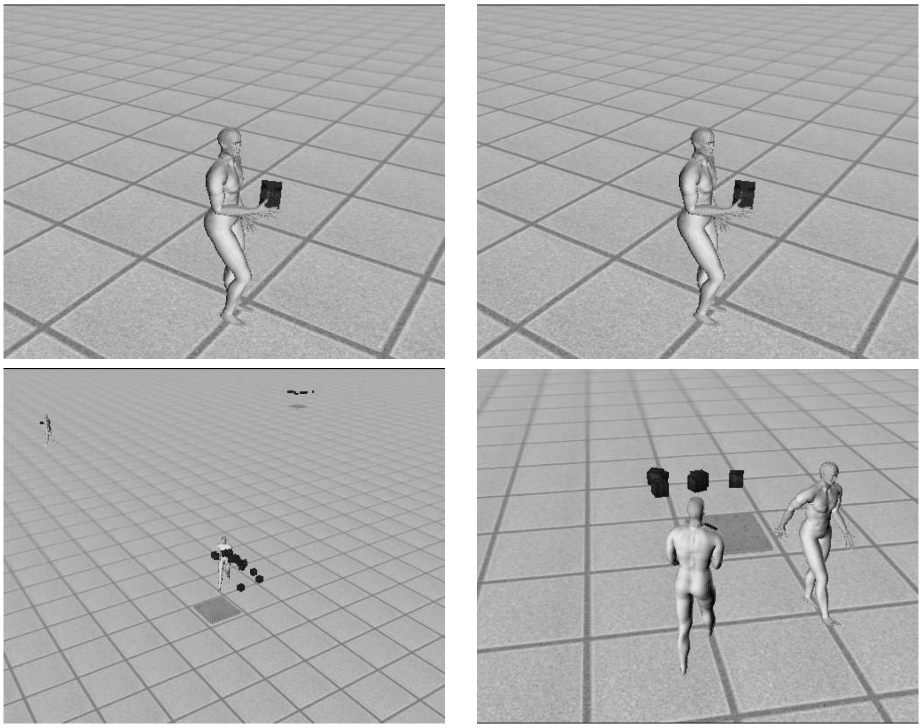

Figure 4: 3D Computer Animation of the Foraging Policy. Upper left and lower left: a bird-eyes view of the animation's beginning and ending. Upper right and lower right: some close-ups.

[7] J. T. Ngo and J. Marks. Spacetime Constraints Revisited. *Proceedings of SIGGRAPH '93*, pp. 343-350, 1993.

[8] C. W. Reynolds. Flocks, Herds and Schools: A Distributed Behavioural Model. *Proceedings of SIGGRAPH '87*, pp. 25-34, 1987.

[9] C. W. Reynolds. Evolution of Corridor Following Behaviour in a Noisy World. From Animals To Animats 3: Proceedings of the Third International Conference on the Simulation of Adaptive Behaviour, pp. 402 - 410. MIT Press, 1994.

[10] R. A. Richards and S. D. Sheppard. Three-dimensional Shape Optimization Utilizing a Learning Classifier System. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (eds.) *Proceedings of the First Annual Conference on Genetic Programming*, July 28-31, 1996, Stanford University; Cambridge, MA: The MIT Press.

[11] G. R. Roberts. Dynamic planning for Classifier Systems. *Proceedings of the Fifth International Conference on Genetic Algorithms*, Morgan Kaufmann, San Mateo, CA, 1993, pp. 231-237.

[12] K. Sims. Evolving 3D Morphology and Behaviour by Competition. *Artificial Life*, v1 n4, pp. 353-372, 1994.

[13] N. Magnenat-Thalmann and D. Thalmann (eds), *Computer Animation '97*, IEEE Computer Society Press, 1997.