GENETIC PROGRAMMING

Riccardo Poli, chair

A re-examination of the Cart Centering problem using the Chorus system

R. Muhammad Atif Azad
Dept. of Computer Science and Information Systems
University of Limerick
Ireland
atif.azad@ul.ie

Conor Ryan
Dept. of Computer Science and Information Systems
University of Limerick
Ireland
conor.ryan@ul.ie

Mark E. Burke
Dept. of Mathematics and Statistics
University of Limerick
Ireland
mark.burke@ul.ie

Ali R. Ansari
Dept. of Mathematics and Statistics
University of Limerick
Ireland
ali.ansari@ul.ie

Abstract

The cart centering problem is well known in the field of evolutionary algorithms and has often been used as a proof of concept problem for techniques such as Genetic Programming. This paper describes the application of a grammar based, position independent encoding scheme, *Chorus*, to the problem. It is shown that using the traditional experimental setup employed to solve the problem, Chorus is able to come up with the solutions which appear to beat the theoretically optimal solution, known and accepted for decades in the field of control theory. However, further investigation into the literature of the relevant area reveals that there is an inherent error in the standard E.C. experimental approach to this problem, leaving room for a multitude of solutions to outperform the apparent best. This argument is validated by the performance of Chorus, producing better solutions at a number of occasions.

1 Introduction

The cart centering problem is a well known problem that often appears in the introductory literature of optimal control (see [Athans, Falb, 66]). In its most basic form, it involves a cart of mass m moving in one dimension on a frictionless horizontal surface. The cart can be moving with any velocity v and can have any position x along the x-axis. The problem is to bring the cart to the origin in a position-velocity space with the values of both x and v approaching zero in minimum amount of time. The literature shows that the problem already has a well defined solution, which guarantees that the cart is centered in minimum amount of time. Genetic programming (GP) [Koza, 92] (pages 122 through 147) has been shown to have successfully solved this problem. The experimental setup described therein shows the absence of any success predicate, meaning that the system is free to wander in the solution space and come up with anything that minimizes the time required to center the cart.

This paper describes the application of a relatively new, position independent, evolutionary automatic programming system, Chorus [Ryan et al, 02a] on this problem. The system involves a genotype phenotype distinction and like [Horner, 96], [Paterson, 97], [Whigham, 95], and Grammatical Evolution (GE) [Ryan, Collins, O'Neill, 98] [O'Neill, Ryan, 01] evolves programs using grammars. While our aim initially for this paper was to demonstrate that Chorus could be successfully applied to the problem, we were surprised to discover that our results showed that the system produced expressions that were able to centre the cart in less time compared to the theoretical optimal control strategy. However a closer examination of the problem, as described in the control genre, reflects that the approach traditionally employed to solve the problem involves an inherent error. As a result there is no unique solution for this problem under the *circumstances*.

The paper first describes a context free grammar in Backus Naur form, which is used to partially specify the behaviour of Chorus, similar to the way in which one specifies **functions** and **terminals** in GP. We then describe the Chorus system and the process involving the mapping from a genotype to phenotype is discussed, with an example. Section 5 describes the application of Chorus on the cart centering problem, the theoretical background, the experimental setup and then discusses the results in the light of literature from control theory. Section 6 draws some conclusions based on the experiences and results presented in the paper.

2 Backus Naur Form

Backus Naur Form (BNF) is a notation for describing grammars. A grammar is represented by a tuple $\{N, T, P, S\}$, where T is a set of terminals, i.e. items that can appear in legal sentences of the grammar, and N is a set of non-terminals, which are interim items used in the generation of terminals. P is the set of production rules that map the non-terminals to the terminals, and S is a start symbol, from which all legal sentences may be generated.

Below is a sample grammar, which is similar to that used by Koza [Koza, 92] in his symbolic regression and integration problems. Although Koza did not employ grammars, the terminals in this grammar are similar to his function and terminal set.

```
S = <expr>

<expr>    ::=  <expr> <op> <expr>   (0)
             | ( <expr> <op> <expr>)(1)
             | <pre-op> ( <expr> )  (2)
             | <var>                (3)
<op>      ::= + (4) | - (5) | % (6)
             | * (7)
<pre-op>  ::= Sin (8) | Cos (9)
             | Exp (A)| Log (B)
<var>     ::= 1.0 (C) | X (D)
```

3 The Chorus System

Chorus[Ryan et al, 02a] is an automatic programming system based coarsely on the manner in which enzymes regulate the model of a cell. Chorus belongs to the same family of algorithms as *Grammatical Evolution* [Ryan, Collins, O'Neill, 98] [O'Neill, Ryan, 01], and shares several characteristics with it. In particular, the output of both systems is governed by a BNF grammar as above, and the genomes, variable length binary strings, interpreted as 8 bit integers (referred to as *codons*), are used to produce legal sentances from the grammar.

There is, however, a crucial difference. It concerns the interpretation of each codon, which, when being processed is **mod**ed with the *total* number of production rules in the grammar. Thus each codon represents a particular production rule, regardless of its position on the chromosome. This behaviour is different from GE, where an integer is **mod**ed with only the number of rules that are *relevant* at that point in time, and the meaning of a codon is determined by those that precede it, leading to the so-called "ripple effect"[Keijzer et al, 01].

For example, consider the individual:

18	28	32	27	42	17	18	31	27	14
45	46	45	18	27	55	65			

which can be looked upon as a collection of hard coded production rules. When moded with the number of rules in the grammar (see section 2), which in this case is 14, the same individual can now be represented as follows (using hexadecimal numbers):

4	0	4	D	0	3	4	3	D	0
3	4	3	4	D	D	9			

Each gene encodes a *protein* which, in our case is a production rule. Proteins in this case are enzymes that regulate the metabolism of the cell. These proteins can combine with other proteins (production rules in our case) to take particular *metabolic pathways*, which are, essentially, phenotypes. The more of a gene that is present in the genome, the greater the *concentration* of the corresponding protein will be during the mapping process [Zubay, 93] [Lewin, 99]. In a coarse model of this, we introduce the notion of a *concentration table*. The concentration table is simply a measure of the concentrations of each of the proteins at any given time, and is initialised with each concentration at zero. At any stage, the protein with the greatest concentration will be chosen, switching on the corresponding metabolic pathway, thus, the switching on of a metabolic pathway corresponds to the development of the forming solution with the application of a production rule.

Many decisions are made during the mapping process. For example, the start symbol `<expr>` has four possible mappings. When such a situation occurs, the relevant area from the concentration table is consulted and the rule with the maximum concentration is chosen. In case there is a tie, or the concentrations of all the rules are zero, the genotype is searched for any of the applicable rules, until a clear winner is found. This is analogous to the scenario where there are a number of voices striving for attention, and only the loudest is heard.

While searching for an applicable production rule, one may encounter rules that are not relevant at that point in time. In this case, the concentrations of those rules are increased, so when that production rule is involved in a decision, it will be more likely to win. This is what brings position independence into the system; the crucial thing is the presence or absence of a gene, while its position is less so. Importantly, *absolute* position almost never matters, while occasionally, *relative* position (to another gene) is important.

Once chosen, the concentration of that production rule is decremented. However, it is not possible for a concentration to fall below zero.

Sticking to the left most non-terminal in the current sentence, mapping continues until there are none left or we are involved in a choice for which there is no concentration either in the table or the genome. An incompletely mapped individual is given a fitness value of exactly zero in the current version of Chorus, thus removing its chances of indulging into any reproductive activity.

3.1 Example Individual

Using the grammar from section 2 we will now demonstrate the genotype-phenotype mapping of a Chorus individual. The particular individual is encoded by the following genome:

18	28	32	27	42	17	18	31	27	14
45	46	45	18	27	55	65			

For clarity, we also show the normalised values of each gene, that is, the genes mod 14. This is only done for readability, as in the Chorus system, the genome is only read on demand, and not decoded until needed.

4	0	4	D	0	3	4	3	D	0
3	4	3	4	D	D	9			

The first step in decoding the individual is the creation of the concentration table. There is one entry for each production rule (0..D), each of which is initially zero. The table is split across two lines to aid readability.

Rule #	0	1	2	3	4	5	6
Concentration							
Rule #	7	8	9	A	B	C	D
Concentration							

The sentence starts as `<expr>`, so the first choice must be made from productions 0..3, that is:

```
<expr>    ::=  <expr> <op> <expr>    (0)
             | ( <expr> <op> <expr>)(1)
             | <pre-op> ( <expr> )   (2)
             | <var>                 (3)
```

None of these have a value yet, so we must read the first gene from the genome, which will cause it to produce its protein. This gene decodes to 4, which is not involved in the current choice. The concentration of 4 is incremented, and another gene read. The next gene is 0, and this is involved in the current choice. Its concentration is amended, and the choice made. As this is the only relevant rule with a positive concentration, it is chosen and its concentration is reduced, and the current expression becomes:

```
<expr><op><expr>
```

The process is repeated for the next leftmost non-terminal, which is another `expr`. In this case, again the concentrations are at their minimal level for the possible choices, so another gene is read and processed. This gene is 4, which is not involved in the current choice, so we move on and keep reading the genome till we find rule 0 which is a relevant rule. Meanwhile we increment the concentrations of rule 4 and D. Similar to the previous step, production rule #0 is is chosen, so the expression is now

```
<expr><op><expr><op><expr>
```

Reading the genome once more for the non-terminal `expr`, produces rule 3 so the expression becomes

```
<var><op><expr><op><expr>
```

The state of the concentration table at the moment is given below.

Rule #	0	1	2	3	4	5	6
Concentration					2		
Rule #	7	8	9	A	B	C	D
Concentration							1

The next choice is between rules #C and #D, however, as at least one of these already has a concentration, the system does not read any more genes from the chromosome, and instead uses the values present. As a result, rule `<var> -> X` is chosen to introduce first terminal symbol in the expression.

Once this non-terminal has been mapped to a terminal, we move to the next left most terminal, `<op>` and carry on from there. If, while reading the genome, we come to the end, and there is still a tie between 2 or more rules, the one that appears first in the concentration table is chosen. However if concentrations of all the relevant rules is zero, the mapping terminates and the individual responsible is given a suitably chastening fitness.

With this particular individual, mapping continues until the individual is completely mapped. The interim choices made by the system are in the order: $4, 3, D, 4, 0, 3, D, 4, 3, D$. The mapped individual is

```
X + X + X + X
```

The state of the concentration table at the end of the mapping is given in the next table.

Notice that there are still some concentrations left in the table. These are simply ignored in the mapping

Rule #	0	1	2	3	4	5	6
Concentration					2		
Rule #	7	8	9	A	B	C	D
Concentration							

process and, in the current version of Chorus, are not used again. Notice also that the rule #9 is not read because the mapping terminates before reading this codon.

4 Genetic Operators

The binary string representation of individuals effectively provides a separation of search and solution spaces. This permits us to use all the standard genetic operators at the string level. Crossover is implemented as a simple, one point affair, the only restriction being that it takes place at the codon boundaries. This is to permit the system to perform crossover on well-formed structures, which promotes the possibility of using schema analysis to examine the propagation of building blocks. Unrestricted crossover will not harm the system, merely make this kind of analysis more difficult.

Mutation is implemented in the normal fashion, with a rate of 0.01, with crossover occuring with a probability of 0.9. Steady state replacement is used, with roulette wheel selection.

As with GE, if an individual fails to map after a complete run through the genome, wrapping operator is used to reuse the genetic material. However, the exact implemenation of this operator has been kept different. Repeated reuse of the same genetic material *effectively* makes a wrapped individual behave like multiple copies of the same genetic material stacked on top of each other in layers. When such an individual is subjected to crossover, the stack is broken into two pieces. When linearized, the resultant of crossover is different from one or the other parent at regular intervals. In order to minimize such happenings, the use of wrapping has been limited to initial generation. After wrapping, the individual is flattened or *unrolled*, by putting all the layers of the stack together in a linear form. The unrolled individual then replaces the original individual in the population. This altered use of wrapping in combination with position flexibility, promises to maintain the exploitative effects of crossover. Unlike GE, the individuals that fail to map on the second and subsequent generations are not wrapped, and are simply considered infeasible individuals.

5 The Cart Centering Problem

The cart centering problem is well known in the area of evolutionary computation. Koza[Koza, 92] successfully applied GP to it, to show that GP was able to come up with a controller that would center the cart in the minimum amount of time possible.

The problem, also referred to as the double integrator problem, appears in introductory optimal control textbooks as the classic application of *Pontryagin*'s Principle (see for instance [Athans, Falb, 66]). There has been considerable research conducted into the theoretical background of the problem, and the theoretical best performance can be calculated, even though designing an expression to produce this performance remains a non-trivial activity.

As Evolutionary Computation methods are bottom up methods, they do not, as such, adhere to problem specific (be it theoretic or practical) information. This means that E.C. can be used as a testing ground for theories - if one can break the barriers proposed by theoreticians, then it probably means that there is a flaw in the theory concerned. However, another possibility is that there is a flaw in the experimental set up, that makes it appear as though the theoretical best has been surpassed.

This section describes the application of Chorus to the cart centering problem, an exercise which ppears to consistently produce individuals that surpass the theoretical best, before discussing the implications of the result.

5.1 Theoretical Background

In its most basic form, we consider a "cart" as a particle of mass m moving in one dimension with position at time t of $x(t)$ relative to the origin, and corresponding velocity $v(t)$. The cart is controlled by an amplitude constrained thrust force $u(t)$, $|u(t)| \leq 1$, and the control objective is to bring the cart to rest at the origin in minimum time on a frictionless track. The state equations are

$$\begin{aligned} \frac{dx}{dt} &= v \\ \frac{dv}{dt} &= \frac{1}{m}u \end{aligned}$$

or

$$\frac{d}{dt}\begin{bmatrix} x \\ v \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} x \\ v \end{bmatrix} + \begin{bmatrix} 0 \\ 1/m \end{bmatrix} u \qquad (1)$$

The solution is a unique "Bang-Bang " control ($u(t)$ takes only the values +1 or -1) with at most 1 switch

which is expressible in feedback form ($u = u^*(x, v)$) in terms of a "switching curve" S in the $x - v$ plane. Following the approach of [Athans, Falb, 66] we find that S is given by

$$x + \frac{m}{2} v|v| = 0, \tag{2}$$

the optimal control by

$$u^* = \begin{cases} -1, & \text{if } x + \frac{m}{2} v|v| > 0 \\ +1, & \text{if } x + \frac{m}{2} v|v| < 0 \\ -v/|v|, & \text{if } x + \frac{m}{2} v|v| = 0 \end{cases} \tag{3}$$

and the minimum time T to reach $(0, 0)$ from (x, v) by

$$T = \begin{cases} mv + \sqrt{2m^2v^2 + 4mx}, & \text{if } x + \frac{m}{2} v|v| > 0 \\ -mv + \sqrt{2m^2v^2 - 4mx}, & \text{if } x + \frac{m}{2} v|v| < 0 \\ m|v|, & \text{if } x + \frac{m}{2} v|v| = 0 \end{cases} \tag{4}$$

The above formulae assume that the system can switch precisely when condition (2) is met. In practice, this is only approximated. The engineering literature contains analyses of what happens when non-ideal switching (deadband and/or hysteresis) occurs using real hardware with the resultant cycling, chattering and steady state error. (see [Gibson, 63] for more details).

5.2 Experimental Setup

GP has been shown to be able to successfully evolve the time optimal control strategy (see [Koza, 92]). The same experimental setup is used by Chorus except where mentioned otherwise. The simulation essentially entails a discretisation of the problem so as to enable a numerical approximation of the derivatives involved. This is referred to as an Euler approximation of the differential equations given in (1), *i.e.,*

$$x(t + h) = x(t) + hv(t),$$

$$v(t + h) = v(t) + \frac{h}{m} u(t),$$

where m is the mass of the cart, h represents the time step size, $v(t + h)$ and $x(t + h)$ represent velocity and distance from the origin respectively at time $t + h$ and $v(t)$ and $x(t)$ represent velocity and distance from the origin respectively at time t. The desired control strategy should satisfy the following conditions.

It should specify the direction of the force to be applied for any given values of $x(t)$ and $v(t)$.

The cart approximately comes to rest at the origin, *i.e.,* the Euclidean (x, v) distance from the origin is less than a certain threshold.

The time required is minimal.

The exact time optimal solution is characterised by the switching condition

$$-x(t) > \frac{v^2(t) Sign\ v(t)}{2|u_{\max}|/m}, \tag{5}$$

which applies the force in the positive x direction if the above condition is met and in the negative direction otherwise. Note that $u_{\max}$ represents the maximum value of $u(t)$, which is 1 here. The $Sign$ function returns +1 for a positive argument and -1 otherwise. For the sake of simplicity m is considered to be equal to 2.0 kilograms and the magnitude of the force $u(t)$ is 1.0 Newtons, so that the denominator equals 1.0 and can be ignored. The experimental settings employed by Koza are summarised in table 1. Note that (5) does not incorporate the equality condition mentioned in (3).

Table 1: A Koza-style Tableau For The Cart Centering Problem.

Objective:	Find a time optimal bang-bang control strategy to center a cart on a one dimensional frictionless track.
Terminal Set:	The state variables of the system: x (position of the cart along X axis), v (velocity V of the cart) and -1.0.
Function Set:	+,-,*,%,ABS,GT.
Fitness cases:	20 initial condition points (x, v) for position and velocity chosen randomly from the square in position-velocity space having opposite corners, $(-0.75, 0.75)$ and $(0.75, -0.75)$.
Fitness:	Reciprocal of sum of the time, over 20 fitness cases, taken to center the cart. When a fitness case times out, the contribution to the sum is 10.0 seconds.
Hits:	Number of fitness cases that did not time out.
Wrapper:	Converts any positive value returned by an expression to +1 and converts all other values (negative or zero) to -1.
Parameters:	M = 500, G = 75
Success Predicate:	None.

The grammar used for the problem is:

```
S = <expr>

<expr>    ::=  <expr> <op> <expr>
             | ( <expr> <op> <expr>)
```

```
                | <pre-op> ( <expr> )
                | <var>
<op>       ::= + | - | % | * | GT
<pre-op>   ::= ABS
<var>      ::= X | V | -1.0
```

The randomly generated 20 fitness cases used by Chorus are given in the table 2.

Table 2: Randomly Generated 20 Starting Points, given as $((x, v)$ pairs).

0.50,0.67	-0.65,0.40	-0.16,-0.57	0.10,0.50
-0.71,0.66	0.43,0.01	-0.28,-0.71	0.27,-0.73
-0.50,0.34	-0.57,0.32	0.43,-0.69	-0.52,-0.16
-0.33,-0.21	-0.16,-0.06	0.71,-0.69	-0.04,-0.63
0.39,0.70	-0.52,-0.42	-0.59,0.38	0.58,-0.35

The cart is considered to be centered if the Euclidean distance from the origin $(0, 0)$ is less than or equal to 0.01. The total time taken by the strategy (5) over all the given set of starting points is 56.07996 seconds. On average it takes 2.803998 seconds per fitness case for the cart to be centered. This means that any strategy which centers the cart in less time, does better than the theoretical solution (5) for this experimental setup.

5.3 Experimental Results

The work of Koza [Koza, 92] shows that the optimal control strategy can be evolved using GP. However, it has not been shown that even in the absence of any success predicate, any strategy was evolved which could beat the result as described by the inequality (5). When the same task is given to the Chorus system, 17 times out of 20 independent runs, it evolves what appears to be a better strategy in terms of time minimisation. Out of those 17 runs, on the average, a better strategy is produced in the 39th generation, the earliest being 20th and the latest being 65th.

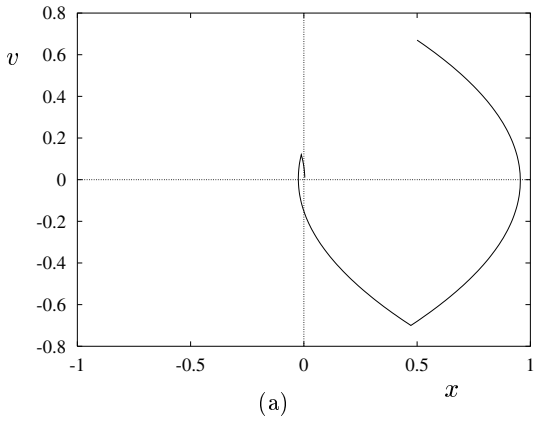
One of the samples which broke the barrier is given as

$$(-1.0 * X)\ GT\ (V * ABS(V) + V * V * V),$$

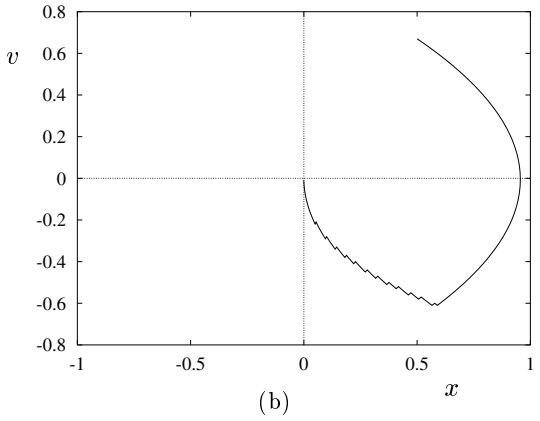
which can be rewritten as

$$-x(t) > v^2(t) Sign\ v(t) + v^3(t), \tag{6}$$

returning $+1$ if the condition is satisfied and -1 otherwise. Total time recorded for this control law mentioned by inequality (6) is 50.799965 seconds over 20 fitness cases which is clearly less than the solution shown by the inequality (5). However, the least time that was recorded was 49.919968 seconds. A plot of x versus v for the control strategy given in (5) is shown in Fig 1(a) for the starting point $(0.50, 0.67)$. A similar plot for the strategy evolved by the Chorus system is shown in Fig 1(b). Notice that in (a) the control strategy crosses the y-axis leading into the negative x-axis region and then it returns to the origin. This shows the longer route traversed by (a) compared to (b) where there is no such occurrence, thus reflecting the time difference between the two strategies.



(a)



(b)

Figure 1: .Trajectories traversed by the two strategies to reach the origin. (a) represents inequality (5) and (b) represents the evolved strategy (6)

5.4 Discussion

It appears from the results in the previous section that a solution better than the theoretical has been achieved. However, a careful consideration of the problem undertaken shows otherwise. This problem has been solved by first discretising the main differential equations as mentioned earlier. The discretisation brings with it an element of error. The time step h

used now plays a major role, in the sense that a smaller time step would lead to a better solution *i.e.,* closer to the theoretical solution (3), and as $h \to 0$ the solution converges to (3).

The time step employed by Koza [Koza, 92] is $h = 0.02$, and using this time step, the error in the derivatives is substantial enough to cause the systems to converge to control laws other than the theoretical result in (3). In this sense Chorus actually validates this by evolving to what is a better solution than (5).

A study of the appropriate literature in the control theory genre indicates that the theoretical model is just that, theoretical. Practical implementation of a control system which brings the cart to the target position is not even "bang-bang" (i.e $u(t)$ is either +1 or -1). Instead, the magnitude of the applied force is any real number between 0 and 1.

One approach is to model the situation as one in which the control can change only at discrete-time steps, either as a sampled data system or a discretised version of eq(1). The former leads to state equations

$$\begin{aligned} x(t+h) &= x(t) + \delta v(t) + \frac{\delta^2}{2m} u(t) \\ v(t+h) &= v(t) + \frac{\delta}{m} u(t) \end{aligned}$$

where $1/\delta$ is the sampling rate. The latter, using an *Euler* discretisation scheme, leads to state equations

$$\begin{aligned} x(t+h) &= x(t) + h v(t) \\ v(t+h) &= v(t) + \frac{h}{m} u(t) \end{aligned}$$

where h is the step size.
When $\delta = h$, both models are of the form

$$\begin{bmatrix} x(t+h) \\ v(t+h) \end{bmatrix} = \begin{bmatrix} 1 & h \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x(t) \\ v(t) \end{bmatrix} + \begin{bmatrix} b \\ h/m \end{bmatrix} u(t) \tag{7}$$

where $b = h^2/2m$ for the sampled data model and $b = 0$ for the discretised model.

The control objective is again to bring the state of eq (7) to the origin in minimum time using a sequence of amplitude constrained controls $|u| \leq 1$. However, due to the discrete time steps, the solution of the problem is fundamentally different to that of the continuous time problem of (1). The optimal control is in general no longer unique, nor except for a set of isolated points in the $x - v$ plane is it Bang-Bang throughout. Hence there are different approaches and algorithms.

The more general problem in n dimensions was initially formulated in [Kalman, 57], and then analysed comprehensively in [Desoer, Wing, 61a] - [Desoer, Wing, 61c]. This analysis, when applied to the cart centering problem recursively constructs a sequence of convex sets $\{C_k\}$, where C_k is the set of states for which there exists an admissible input sequence which transfers the state to the origin in k time steps but no fewer ($C_0 = \{(0,0)\}$). For instance, if we want to centre the cart in 1 time step then C_1 represents the region of interest. For any $(x_1, v_1) \in C_1$, the cart is guaranteed to be centered in exactly 1 time step. In addition, a piecewise linear switching curve is constructed which divides the plane into regions of positive and negative control values (see figure 2).

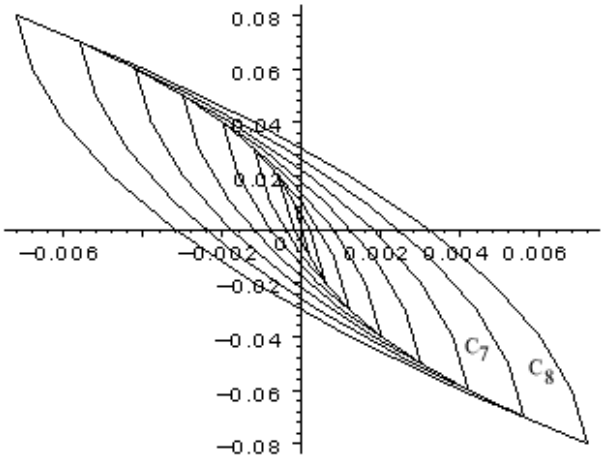


Figure 2: The sets C_1 - C_8 for the *Euler* discretised system with a cart of mass $m = 2$ and $h = 0.02$

Later work has looked at describing the C_k in terms of their facets with associated algorithms [Keerthi, Gilbert, 87], and there is still much interest in improving the efficiency of the existing algorithms (see [Jamak, 00] for a good review).

6 Conclusions

We have described the application of a position independent, representation scheme for Evolutionary Algorithms, termed Chorus, on the cart centering problem. Much to our surprise, Chorus apparently succeeded in producing individuals that performed better than the theoretical best. However, further analysis of the problem and traditional experimental set up revealed flaws that changed the nature of the problem.

The paper describes how Chorus was able to exploit these flaws to produce surprisingly fit individuals, and how an Evolutionary Computation system can be used to help test models of physical systems. Also, it re-emphasizes the point that while attempting to solve continuous problems numerically, we should be ware of the resultant discretisation errors. It is also worth noting that the way these problems are typically solved by control engineers is by starting with the discretised analogues of the continuous problems and then proceeding to solve. It might be worth exploring what a system like Chorus may have to offer in the solution process of such a discretised problem.

6.1 Future Work

The results shown in the cart centering problem encourage the use of Chorus for real world problems. Coupled with the strengths of the system discussed in [Ryan et al, 02a], the system can be applied but not limited to the problems in the field of control theory and fluid dynamics.

Chorus diverges considerably from algorithms in the same "family", e.g. Grammatical Evolution and GAUGE[Ryan et al, 02b] in that it does not exploit the ripple effect, and instead uses position independent, absolute genes. This makes Chorus very suitable for schema analysis, and also possible that the Genetic Algorithm Schema Theory could, with very little extension, be applied to an automatic programming system.

References

[Athans, Falb, 66] Athans M. and P. L. Falb, *Optimal Control*, McGraw-Hill, 1966.

[Desoer, Wing, 61a] Desoer C.A. and J. Wing, "An Optimal Strategy for a Saturating Sampled-Data System", *IRE Transactions on Automatic Control*, vol AC-6, pp. 5-15, 1961.

[Desoer, Wing, 61b] Desoer C.A. and J. Wing, "A Minimal Time Discrete System", *IRE Transactions on Automatic Control*, vol AC-6, pp. 111-125, 1961.

[Desoer, Wing, 61c] Desoer C.A. and J. Wing, "Minimal Time Regulator Problem for Linear Sampled-Data Systems (General Theory)", *J. Franklin Inst.*, vol 20, pp. 208-228, 1961.

[Gibson, 63] Gibson J.E., *Nonlinear Automatic Control*, McGraw-Hill, 1963.

[Goldberg, Korb, Deb, 89] Goldberg D E, Korb B, Deb K. Messy genetic algorithms: motivation, analysis, and first results. *Complex Syst. 3*

[Horner, 96] H. Horner, *A C++ class library for Genetic Programming: The Vienna University of Economics Genetic Programming Kernel.* Release 1.0, Operating instruction. Vienna University of Economics, 1996.

[Jamak, 00] Jamak A., "Stabilization of Discrete-Time Systems with Bounded Control Input", MASc Dissertation, University of Waterloo, Waterloo CA.,2000.

[Kalman, 57] Kalman, R.E., "Optimal Nonlinear Control of Saturating Systems by Intermittent Action" , 1957 IRE Wescon Convention Record, pt 4, pp 130-135.

[Keerthi, Gilbert, 87] Keerthi S.S. and E.G. Gilbert, "Computation of Minimum-Time Feedback Control Laws for Discrete-Time Systems with State-Control Constraints", *IEEE Transactions on Automatic Control*, vol AC-32, pp. 432-435, 1987.

[Keijzer et al, 01] Keijzer M., Ryan C., O'Neill M., Cattolico M., Babovic V. Ripple Crossover In Genetic Programming, in *Proceedings of EuroGP'2001.*

[Koza, 92] J. Koza. "Genetic Programming". MIT Press, 1992.

[Lewin, 99] Lewin B. *Genes VII.* Oxford University Press, 1999.

[O'Neill, Ryan, 01] O'Neill M., Ryan C. Grammatical Evolution. *IEEE Transactions on Evolutionary Computation.* 2001.

[Paterson, 97] N. Paterson and M. Livesey, "Evolving caching algorithms in C by GP" in *Genetic Programming 1997: Proc. 2nd Annu. Conf.*, MIT Press, 1997, pp. 262-267. MIT Press.

[Ryan, Collins, O'Neill, 98] C. Ryan, J.J. Collins and M. O'Neill, "Grammatical Evolution: Evolving Programs for an Arbitrary Language", in *EuroGP'98: Proc. of the First European Workshop on Genetic Programming* (Lecture Notes in Computer Science 1391), Paris, France: Springer 1998, pp. 83-95.

[Ryan et al, 02a] C. Ryan, A. Azad, A. Sheahan and M. O'Neill, "No Coercion and No Prohibition, A Position Independent Encoding Scheme for Evolutionary Algorithms - The Chorus System". In the *Proceedings of European Conference on Genetic Programming (EuroGP 2002).*

[Ryan et al, 02b] C. Ryan, Miguel Nicolau, and M. O'Neill, "Genetic Algorithms Using Grammatical Evolution". In the *Proceedings of European Conference on Genetic Programming (EuroGP 2002).*

[Whigham, 95] P. Whigham, "Grammatically-based Genetic Programming" in *Proceedings of the Workshop on GP: From Theory to Real-World Applications*, Morgan Kaufmann, 1995, pp. 33-41.

[Zubay, 93] Zubay G. Biochemistry. Wm. C. Brown Publishers, 1993

A Survey and Analysis of Diversity Measures in Genetic Programming

Edmund Burke **Steven Gustafson**[†] **Graham Kendall**
School of Computer Science & IT
University of Nottingham
Nottingham, UK NG81BB
{ekb | smg | gxk}@cs.nott.ac.uk
[†]corresponding author

Abstract

This paper presents a survey and comparison of the significant diversity measures in the genetic programming literature. The overall aim and motivation behind this study is to attempt to gain a deeper understanding of genetic programming dynamics and the conditions under which genetic programming works well. Three benchmark problems (Artificial Ant, Symbolic Regression and Even-5-parity) are used to illustrate different diversity measures and to analyse their correlation with performance. The results show that diversity is not an absolute indicator of performance and that phenotypic measures appear superior to genotypic ones. Finally we conclude that interesting potential exists with tracking ancestral lineages.

1 INTRODUCTION

Maintaining population diversity in genetic programming (Banzhaf et al., 1998) is referred to as the key in preventing premature convergence and stagnation in local optima (McPhee and Hopper, 1999)(Ryan, 1994)(Ekárt and Németh, 2000)(McKay, 2000)(Rosca, 1995a). Diversity is the amount of variety in the population defined by what genetic programming individuals 'look' like or how they 'perform'. The number of different fitness values (phenotypes) (Rosca, 1995b), different structural individuals (genotypes) (Langdon, 1996), edit distances between individuals (Ekárt and Németh, 2000), and complex and composite measures (McKay and Abbass, 2001)(Keijzer, 1996)(D'haeseleer, 1994) are used as measures of diversity. At the individual level, diversity measures differences between individuals and is used to select individuals for reproduction or replacement (Eshelman and Schaffer, 1993).

In this study, we examine the previous uses and meanings of diversity, compare these different measures on three benchmark problems and discuss the results. As far as the authors are aware, all the significant diversity measures that occur in the genetic programming literature are reported.

The ultimate goal is to determine a good measurement of population diversity and understand the effects of its influence as the evolutionary search progresses. The overall motivation of this study is that a better understanding of diversity and diversity measures will lead to a better understanding of genetic programming and the advantages and disadvantages of employing it in any given situation.

The following sections examine different measures of diversity, how these measures relate to each other and how they relate to the performance of three genetic programming problems. Section 2 describes measures of population diversity and previous methods of promoting diversity in populations. Section 3 describes the experiments. Section 4 presents and discuss results. Section 5 draws some brief conclusions and Section 6 outlines some ideas for future work.

2 DIVERSITY MEASURES

Some measures of diversity are intended to quantify the variety in a population and others are used to measure the difference between individuals. The latter type is used to attempt to control or promote high diversity during a run. The following section surveys both measures that provide a quantification of population diversity and methods used to actively promote and maintain diversity within genetic programming.

2.1 POPULATION MEASURES

The most common type of diversity measure is that of structural differences between programs. Koza (1992) used the term *variety* to indicate the number of different programs his populations contained. In this measure, two programs are structurally compared, looking for exact matches. Landgon (1996) felt that genotypic diversity was a sufficient upper bound of population diversity as a decrease in unique individuals must also mean a decrease in unique fitness values. More complex genotype measures count subtrees, size, and type and frequencies of nodes.

Keijzer (1996) measures program variety by the number of unique individuals and subtree variety by counting unique subtrees. Population diversity is a ratio of the number of unique individuals over population size and subtree variety is the ratio of unique subtrees over total subtrees. Tackett (1994) also measures structural diversity using unique subtrees and schemata frequencies. D'haeseleer and Bluming (1994) define the frequency of terminals and functions as "genotypical diversity" and fitness case results as "phenotypical diversity", which are correlated within the population for their study of local populations and demes.

When tree representations of genetic programs are considered as graphs, individuals can be compared for isomorphism (Rosca, 1995a) to obtain a more accurate measure of diversity. Determining graph isomorphism, however, is computationally expensive for an entire population. We could count the number of nodes, terminals, functions and other graph properties in a reasonable time and use this n-tuple to determine whether trees are *possible* isomorphs of each other.

McPhee and Hopper (1999) investigate diversity at the genetic level by tagging each node created in the initial generation with a unique *id*. Root parents, the parents whose tree has a portion of another individual's subtree swapped into it during crossover, are assigned new *memid*s, an auxiliary tag that is initially the same value of the *id*. All the nodes from the root down to the crossover point are assigned new *memid*s to indicate that these nodes have one new child. If there is no mutation in the genetic programming system (as here), then no new *id*s will be created after the initial generation, only *memid*s. McPhee and Hopper found that the number of unique *id*s dramatically falls after initial generations and, by tracking the root parents, after an average of 16 generations, all further individuals have the same common root ancestor.

Phenotypic measures compare the number of unique fitness values in a population. When the genetic programming search is compared to traversing a fitness landscape, this measure provides an intuitive way to think of how the population covers that landscape. Other measures could be created by using fitness values of a population, as done by Rosca (1995a) with entropy and free energy. Entropy here represents the amount of disorder of the population, where an increase in entropy represents an increase in diversity. Bersano-Begey (1997) track how many individuals solve which fitness cases. By monitoring the population, a pressure is added to individuals to promote the discovery of different or less popular solutions.

2.2 PROMOTING DIVERSITY

Several measures and methods have been used to promote diversity by measuring the difference between individuals. These methods typically use a non-standard selection, mating, or replacement strategy to bolster diversity. Neighborhoods, islands, niches, crowding and sharing from genetic algorithms are common themes to these methods.

Eschelman and Schaffer (1993) use Hamming distances between individuals to select individuals for recombination and replacement to improve over hill-climbing-type selection strategies for genetic algorithms.

Ryan's (1994) "Pygmie" algorithm builds two lists based on fitness and length to facilitate selection for reproduction. The algorithm maintains more diversity and prevents premature convergence. The advantage of this algorithm is that it does not attempt to "over-control" evolution and uses simple measures to promote diversity.

De Jong et al (2001) use multiobjective optimisation to promote diversity and concentrate on non-dominated individuals according to a 3-tuple of <*fitness,size,diversity*>. Diversity is the average square distance to other members of the population, using a specialised measure of edit distance between nodes. This multiobjective method promotes smaller and more diverse trees.

McKay (2000) applies the traditional fitness sharing concept from Deb and Goldberg (1989) to test its feasibility in genetic programming. Diversity is the number of fitness cases found, and the sharing concept assigns a fitness based on an individual's performance divided by the number of other individuals with the same performance. McKay also studies negative correlation and a *root quartic negative correlation* in (2001) to preserve diversity. Ekárt and Németh (2000) apply fitness sharing with a novel tree distance definition and suggest that it may be an efficient measure of struc-

tural diversity.

By surveying previous work using diversity measures, we designed several experiments to determine relationships between different population measures of diversity and how they correlate to the best fitness of a run.

3 EXPERIMENTS

In this study we would like to answer two questions: One, how do different measures of diversity relate to each other, and two, how do those measures correlate to the best fitness of a run. Three common problems are used with common parameter values from previous studies. For all problems, a population size of 500 individuals, a maximum depth of 10 for each individual, a maximum depth of 4 for the tree generation half-n-half algorithm and internal node selection probability of 0.9 for crossover is used. Additionally, each run consists of 51 generations, or until the ideal fitness is found.

The Artificial Ant, Regression and Even-5-Parity problems are used. All three problems are typical to genetic programming and can be found in many studies, including (Koza, 1992). The ant problem is concerned with finding the best strategy for picking up pellets along a trail in a grid. The fitness for this problem is measured as the number of pellets missed. The regression problem attempts to fit a curve for the function $x^4+x^3+x^2+x$. Fitness here is determined by summing the squared difference for each point along the objective function and the function produced by the individual. The parity problem takes an input of a random string of 0's and 1's and outputs whether there are an even number of 1's. The even-5-parity fitness is the number of wrong guesses for the 2^5 combinations of 5-bit length strings.

To produce a variety of run performances, where we consider the best fitness in the last generation, we designed three different experiments, carried out 50 times, for each problem. The first experiment *random* performs 50 independent runs. The experiment *stepped-recombination* does 50 runs with the same random number seed, where each run uses an increasing probability for reproduction and decreasing probability for crossover. Initially, probability for crossover is 1.0, and this is decreased by 0.02 each time, skipping the value of reproduction set to .98 to allow for exactly 50 runs and ending with reproduction probability of 1.0 and crossover probability 0.0. The last experiment *stepped-tournament* is similar but we begin with a tournament size of 1 and increment this by 1 for each run, until we reach a tournament size of 50. In the *random* and *stepped-tournament* experiments, crossover probability is set to 1.0 and the tournament size in *random* and *stepped-recombination* is 7. The *Evolutionary Computation in Java* (ECJ), version 7.0, (Luke, 2002) is used, where each problem is available in the distribution.

In analysing the results, we compare the 50 runs for fluctuations of diversity levels in the different measures and examine the standard deviation across experiments for each problem. Additionally, the Spearman correlation coefficient (Siegel, 1956) is computed, comparing the ranking of a run's performance and diversity measure for that run (also taken from the last generation's population).

The following measures of diversity were introduced previously and are described next as they are collected for each generation in every run.

Unique Node *id*: Tag each node with *id:memid* as in (McPhee and Hopper, 1999) and count number of distinct *id*s in each generation.

Size of Ancestral Pool: Since each individual has one root ancestor, in any generation each individuals' line of root ancestors can be traced to the initial generation. It is possible to consider the size of the set that is formed by a set of root parents from the initial generation, and then replacing this set with its intersection with the next generation's root parents. A common ancestor exists when the size becomes 1.

Entropy: Calculate the entropy of the population as in (Rosca, 1995a). Entropy is represented as, where "p_k is the proportion of the population P occupied by population partition k":

$$-\sum_k p_k \cdot \log p_k$$

Here a partition is assumed to be each possible different fitness value, but could be defined to include a subset of values.

Pseudo-Isomorphs: Calculate pseudo isomorphs by defining a 3-tuple of <terminals,nonterminals,depth>, for each individual and count the number of unique 3-tuples in each population. Two identical 3-tuples represent trees which could be isomorphic.

Genotypes and Phenotypes: Count the number of unique trees for the genotype measure (Langdon, 1996). The number of unique fitness values in a population represents the phenotype measure (Rosca, 1995b).

The Spearman correlation coefficient is computed as

follows (Siegel, 1956):

$$1 - \frac{6 \sum_{i=1}^{N} d_i^2}{N^3 - N}$$

Where N is the number of items (50 runs), and d_i is the distance between each run's rank of performance and rank of diversity in the last generation. A value of -1.0 represents negative correlation, 0.0 is no correlation and 1.0 is positive correlation. For our measures, if we see ideal low fitness values, which will be ranked in ascending order (1=best,...,50=worst) and high diversity, ranked where (1=lowest diversity and 50=highest diversity), then the correlation coefficient should be strongly negative. Alternatively, a positive correlation indicates that either bad fitness accompanies high diversity or good fitness accompanies low diversity.

4 RESULTS AND DISCUSSION

Graphs of 50 runs for each of the three experiments and each problem were examined. Graphs for the ant and regression problems are shown in Figures 1-4. The min, max and standard deviation of each measure (including best fitness) were calculated for each run and the Spearman correlation coefficient was calculated for each of the six diversity measures versus run performance, found in Table 1. This study involved 450 runs of 51 generations each, with each population consisting of 500 individuals, or 13,500,000 individual evaluations.

We found relatively stable standard deviations of best fitness in the ant problem experiments (11.8575, 12.9049, 12.0785) but there were large difference in standard deviations of genotype diversity (14.4554, 124.7823, 37.3990). This variation in best fitness is not indicated by the number of unique trees (genotypes): There is a minimum value of 428 and a maximum of 489. This consistently high genotype diversity does not suggest a strong relationship with the varying performance.

Unique node *ids* and root ancestors converge early in each run. This confirms the results found in (McPhee and Hopper, 1999) that genetic-level diversity is lost very quickly, even with widely varied performance, recombination and tournament values. A further study to consider when these measures converge could be an interesting indicator of other diversity or run performance values. In nearly all of the graphs of diversity measures and best fitness, the most dramatic activity occurs when the number of unique *ids* and root ancestors converges. This activity can been seen in

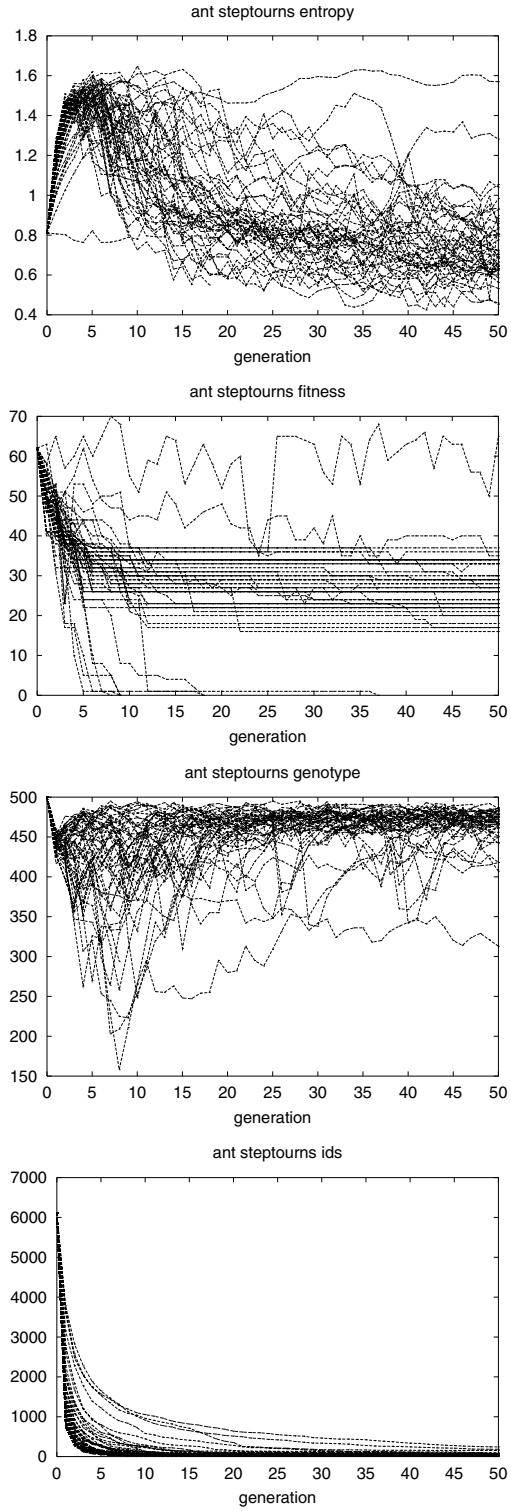


Figure 1: 50 runs of best fitness per generation (top graph) for the ant stepped-tournament experiment. Here, low fitness is better. Also a graph for each of the diversity measures of entropy, genotype, unique node *ids*.

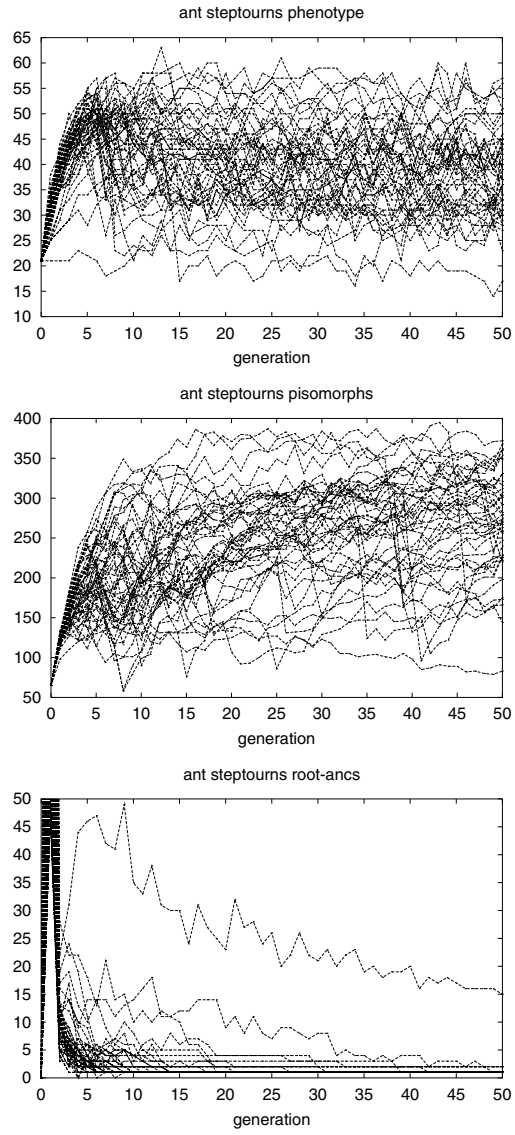


Figure 2: 50 runs of the ant stepped-tournament experiments, showing a graph for each of the diversity measures of phenotype, pseudo-isomorphs, and root ancestors.

Figures 1 through 4. It is not clear, however, how this phenomenon effects evolution and loss of diversity (according to other measures) since, when the number of unique *ids* is reduced and even when a common root ancestor is found, runs are still capable of finding good solutions.

Using the Spearman correlation coefficient we investigated whether runs that produced good fitness had low/high diversity, where ties in ranks were resolved by splitting the rank among tying items (add possible ranks and average). Remembering that negative

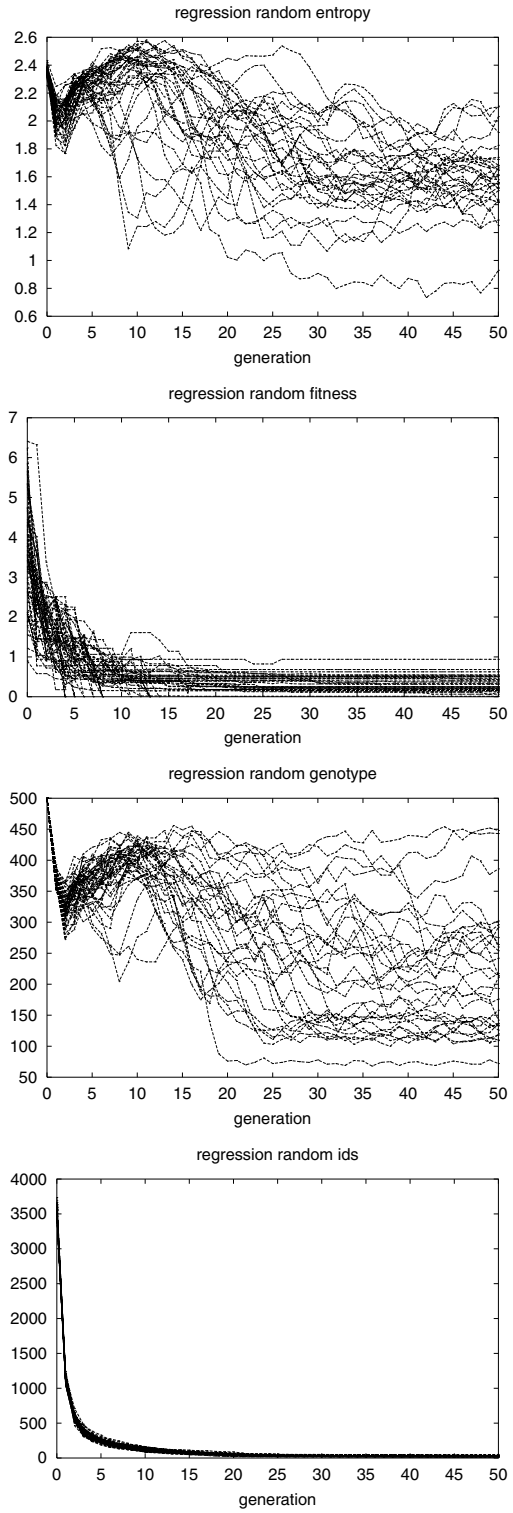


Figure 3: 50 runs of best fitness per generation (top graph) for the regression random experiment. Here, low fitness is better. Also a graph for each of the diversity measures of entropy, genotype, unique node *ids*.

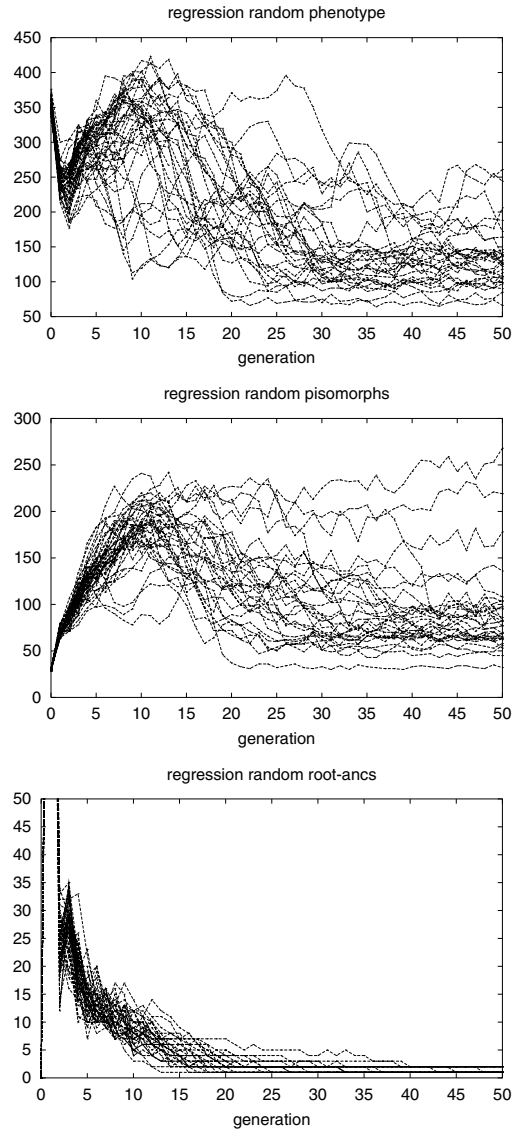


Figure 4: 50 runs of the ant stepped-tournament experiments, showing a graph for each of the diversity measures of phenotype, pseudo-isomorphs, and root ancestors.

correlation (values close to -1.0) suggest that high diversity is correlated with good performance. Table 1 provides the data for all experiments. High negative correlation is seen most consistently with entropy and phenotype diversity. Genotype diversity showed high negative correlation on the regression problem but otherwise varied between little to positive correlation on other problems. While phenotype and entropy always had a negative correlation with performance, values ranged from -0.1608 to -0.8893 with an average correlation of -0.6019 for phenotype and -0.6054 for entropy diversity across all experiments. These were the only

Table 1: Problems ant (a), regression (r) and parity (p) with experiments *random* (rand), *stepped-tournament* (step-t) and *stepped-recombination* (step-r). Values are from the final population. Best fitness ("b.fit") is the best fitness in the final generation. The Spearman correlation coefficient shows perfect correlation with value 1.0 and perfect negative correlation with value -1.0.

prob.	expr.	col.	spearman	min max	stand.dev
a	rand	b.fit		0.0 39.0	11.8575
a	rand	ids	0.1727	25.0 145.0	22.4092
a	rand	roots	0.5014	1.0 1.0	0.0
a	rand	phene	-0.1608	16.0 59.0	8.0181
a	rand	gene	0.4081	428.0 489.0	14.5543
a	rand	isom	0.5391	121.0 350.0	63.3594
a	rand	entro	-0.4195	0.4215 1.1566	0.1702
a	step-r	b.fit		0.0 62.0	12.9049
a	step-r	ids	0.0155	15.0 110.0	24.1658
a	step-r	roots	0.1740	1.0 4.0	0.5291
a	step-r	phene	-0.4088	1.0 47.0	9.6260
a	step-r	gene	0.0799	1.0 477.0	124.7823
a	step-r	isom	0.3532	1.0 348.0	83.0020
a	step-r	entro	-0.5590	-0.0 1.1457	0.2160
a	step-t	b.fit		0.0 65.0	12.0785
a	step-t	ids	0.2351	14.0 242.0	42.4240
a	step-t	roots	0.4253	1.0 15.0	1.9673
a	step-t	phene	-0.2854	17.0 57.0	8.9314
a	step-t	gene	0.3040	294.0 488.0	37.3990
a	step-t	isom	0.3394	83.0 372.0	67.0498
a	step-t	entro	-0.3461	0.4525 1.5702	0.2155
r	rand	b.fit		0.0 0.9399	0.2310
r	rand	ids	-0.6552	16.0 342.0	89.9100
r	rand	roots	-0.6393	1.0 21.0	5.3113
r	rand	phene	-0.7159	66.0 377.0	95.6887
r	rand	gene	-0.5779	72.0 448.0	114.2444
r	rand	isom	-0.5321	32.0 268.0	53.2196
r	rand	entro	-0.6882	0.9297 2.5029	0.4044
r	step-r	b.fit		0.0 2.8999	0.4552
r	step-r	ids	-0.5228	4.0 99.0	14.9947
r	step-r	roots	0.0244	1.0 8.0	1.5133
r	step-r	phene	-0.8703	1.0 303.0	61.0422
r	step-r	gene	-0.8318	1.0 347.0	76.7983
r	step-r	isom	-0.8082	1.0 165.0	36.1054
r	step-r	entro	-0.8430	-0.0 2.2878	0.4713
r	step-t	b.fit		0.0 2.8999	0.4338
r	step-t	ids	-0.5199	8.0 208.0	39.7216
r	step-t	roots	-0.0021	1.0 16.0	3.3859
r	step-t	phene	-0.5797	22.0 428.0	88.6046
r	step-t	gene	-0.5043	28.0 458.0	108.1168
r	step-t	isom	-0.4479	17.0 249.0	49.4191
r	step-t	entro	-0.4001	1.0748 2.5894	0.3214
p	rand	b.fit		3.0 12.0	1.9267
p	rand	ids	-0.0142	29.0 93.0	12.6820
p	rand	roots	0.5189	1.0 1.0	0.0
p	rand	phene	-0.6950	7.0 16.0	1.9489
p	rand	gene	0.2001	422.0 478.0	14.2580
p	rand	isom	0.2635	46.0 92.0	11.5526
p	rand	entro	-0.6777	0.5138 0.9241	0.08773
p	step-r	b.fit		5.0 14.0	2.1462
p	step-r	ids	-0.4573	15.0 57.0	12.4997
p	step-r	roots	0.5119	1.0 1.0	0.0
p	step-r	phene	-0.8119	1.0 13.0	2.4278
p	step-r	gene	-0.5957	1.0 471.0	103.8743
p	step-r	isom	-0.0526	1.0 111.0	18.3605
p	step-r	entro	-0.7039	-0.0 0.8291	0.1801
p	step-t	b.fit		1.0 15.0	2.6510
p	step-t	ids	0.2629	20.0 225.0	32.6593
p	step-t	roots	0.5934	1.0 16.0	2.1344
p	step-t	phene	-0.8893	3.0 17.0	2.5258
p	step-t	gene	0.4247	344.0 485.0	28.9229
p	step-t	isom	0.2311	39.0 102.0	13.9385
p	step-t	entro	-0.8115	0.0445 0.9432	0.1325

measures that did not show some positive correlation.

Correlation values were not consistently high (statistical significant) but indicate that a relationship may be present. Scatter plots show trends indicated by the Spearman correlation, and Figure 5 shows plots for

the regression problem and *stepped-recombination* experiment. Notice the obvious correlation between low fitness rankings and high diversity rankings for each of the 50 runs for the phenotype, genotype, pseudo-isomorphs and entropy measures. Results suggest that one measure is not definitive but different measures may provide useful information for different problems.

The appearance of consistent negative correlations suggests that better performing runs do have higher diversity. Also confirmed by the correlation study is that the entropy and phenotype measures, and the genotype and pseudo-isomorph measures each have similar results. Since phenotype and pseudo-isomorphs would seem to be less computationally expensive, these measures may be more desirable to track in evolutionary computation systems.

5 CONCLUSIONS

The measures of diversity surveyed and compared here demonstrate that the typical genotype measure may not be sufficient to accurately capture the dynamics of a population, which is also suggested in (Ryan, 1994)(Keijzer, 1996).

High variance in performance was not indicated by genotype diversity. The phenotype and entropy measures appear to correlate better with run performance and are less expensive to compute.

The pseudo-isomorph measure appeared to be as informative as genotype diversity and suggests that this simpler measure may be more desirable. Additionally, the consistent early convergence of unique node *ids* and root ancestors, coupled with significant activity in the other measures and performance, show interesting potential for more study.

The relationship between diversity and run performance is not straightforward, and our results indicated some measures had a stronger correlation than others, but not in all experiments and in all problems. This study illustrates the need to carefully define diversity and consider the effects of problem and fitness representation.

6 FUTURE WORK

Several extensions to this research were identified and are currently underway. Further experiments on more problems (including real-world) will provide a more thorough investigation. By tracking the convergence of unique *id*s, root ancestors and other measures during evolution, it is hoped that an early indicator for run

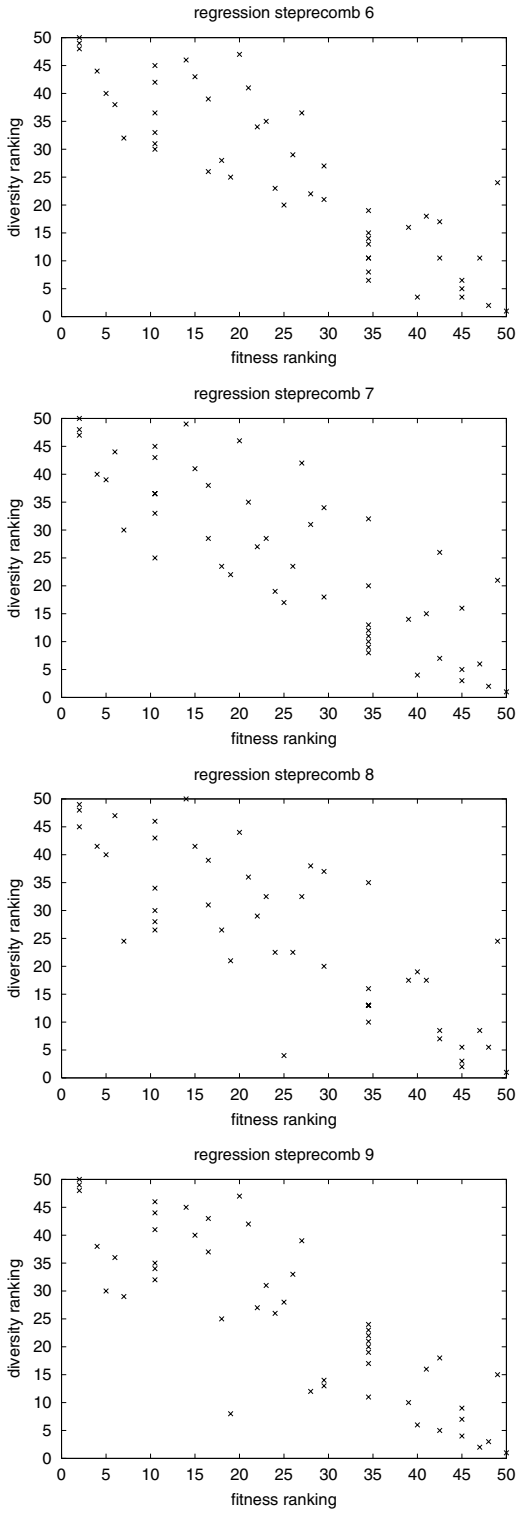


Figure 5: Scatter plots of diversity measures (6=phenotype, 7=genotype, 8=pseudo-isomorphs, 9=entropy) versus best fitness from last generation.

success or failure can be found. Also of interest is using methods to promote diversity and then applying these different diversity measures to determine their effects of improving diversity. Several different and novel diversity measures are also being investigated. The last item of current work examines the computation needed for maintaining the most efficient knowledge (of the evolutionary computation system) to determine effective diversity measures. The research reported is being extended and early experiments indicate that diversity measures based on edit distances provide complimentary and interesting results.

Acknowledgments

The authors would like to thank Natalio Krasnogor, David Gustafson, J. Dario Landa Silva and conference reviewers for comments on this work.

References

Banzhaf, W., Nordin, P., Keller, R. E., and Francone, F. D. 1998. *Genetic Programming: An Introduction*. Morgan Kaufmann, Inc., San Francisco, USA.

Bersano-Begey, T. 1997. Controlling exploration, diversity and escaping local optima in GP: Adapting weights of training sets to model resource consumption. In J. Koza (ed.), *Late Breaking Papers at the 1997 Genetic Programming Conference*, Stanford University, CA.

de Jong, E., Watson, R., and Pollack, J. 2001. Reducing bloat and promoting diversity using multi-objective methods. In L. Spector et al. (eds.), *Proceedings of the Genetic and Evolutionary Computation Conference*, Morgan Kaufmann, San Francisco, CA.

Deb, K. and Goldberg, D. 1989. An investigation of niche and species formation in genetic function optimization. In J. D. Schaffer (ed.), *Proceedings of the Third International Conference on Genetic Algorithms*, pp 42–50, Washington DC, USA.

D'haeseleer, P. 1994. Context preserving crossover in genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, Vol. 1, pp 256–261, IEEE Press, Orlando, FL, USA.

Ekárt, A. and Németh, S. Z. 2000. A metric for genetic programs and fitness sharing. In R. Poli et al. (eds.), *Proceedings of the European Conference on Genetic Programming*, Vol. 1802 of *LNCS*, pp 259–270, Springer-Verlag, Edinburgh.

Eshelman, L. and Schaffer, J. 1993. Crossover's niche. In S. Forrest (ed.), *Proceedings of the Fifth International Conference on Genetic Algorithms*, pp 9–14, Morgan Kaufman, San Mateo, CA.

Keijzer, M. 1996. Efficiently representing populations in genetic programming. In P. Angeline and K. Kinnear, Jr. (eds.), *Advances in Genetic Programming 2*, Chapt. 13, pp 259–278, MIT Press, Cambridge, MA, USA.

Koza, J. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA.

Langdon, W. 1996. *Evolution of Genetic Programming Populations*. Research Note RN/96/125, University College London, Gower Street, London WC1E 6BT, UK.

Luke, S. 2002. *ECJ: A Java-based evolutionary comp-utation and genetic programming system*. http://www.cs.umd.edu/projects/plus/ecj/.

McKay, R. 2000. Fitness sharing in genetic programming. In D. Whitley et al. (eds.), *Proceedings of the Genetic and Evolutionary Computation Conference*, pp 435–442, Morgan Kaufmann, Las Vegas, Nevada, USA.

McKay, R. and Abbass, H. 2001. Anticorrelation measures in genetic programming. In *Australasia-Japan Workshop on Intelligent and Evolutionary Systems*.

McPhee, N. and Hopper, N. 1999. Analysis of genetic diversity through population history. In W. Banzhaf et al. (eds.), *Proceedings of the Genetic and Evolutionary Computation Conference*, Vol. 2, pp 1112–1120, Morgan Kaufmann, Florida, USA.

Rosca, J. 1995a. Entropy-driven adaptive representation. In J. Rosca (ed.), *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pp 23–32, Tahoe City, California, USA.

Rosca, J. 1995b. Genetic programming exploratory power and the discovery of functions. In J. McDonnell et al. (eds.), *Proceedings of the Fourth Annual Conference on Evolutionary Programming*, pp 719–736, MIT Press, San Diego, CA, USA.

Ryan, C. 1994. Pygmies and civil servants. In K. Kinnear, Jr. (ed.), *Advances in Genetic Programming*, Chapt. 11, pp 243–263, MIT Press.

Siegel, S. 1956. *Nonparametric Statistics for the Behavioral Sciences*. McGraw-Hill Book Company, Inc.

Tackett, W. 1994. *Ph.D. thesis*, University of Southern California, Department of Electrical Engineering Systems, USA.

Fitness Distance Correlation and Problem Difficulty for Genetic Programming

Manuel Clergue
I3S Laboratory,
University of Nice,
Sophia Antipolis, France

Philippe Collard
I3S Laboratory,
University of Nice,
Sophia Antipolis, France

Marco Tomassini
Computer Science Inst.,
University of Lausanne,
Lausanne, Switzerland

Leonardo Vanneschi
Computer Science Inst.,
University of Lausanne,
Lausanne, Switzerland

Abstract

This work is a first step in the attempt to verify whether (and in which cases) fitness distance correlation can be a good tool for classifying problems on the basis of their difficulty for genetic programming. By analogy with the studies that have already been done on genetic algorithms, we define some notions of distance between genotypes. Then we choose one of these distances to calculate the fitness distance correlation coefficient and we use it to study the difficulty of some problems. First, we do this for a syntactically limited language. Then we extend the study to standard genetic programming. For the functions used here i.e., traps and royal trees, the results confirm that fitness distance correlation is a good predictor of genetic programming difficulty.

1 INTRODUCTION

Previous studies of problem difficulty in Genetic Algorithms (GAs) [6] have shown the importance of the notion of *distance* between genotypes as a tool to measure the difficulty of problems. The typical distance definition for GAs is the Hamming distance, while other distances like alternation [3] have also been used. Here we present for the first time a study for GP difficulty in analogy with GA research. For this reason, in the following, we offer some definitions of distance between individuals for GP. It should be noted that fitness landscape structure depends on the operators used to traverse it [6]. In the present work, the standard GP crossover [10] is used as the sole genetic operator.

The usual approach to problem difficulty in GP has been the use of a more or less agreed upon set of test problems that have their origin in Koza's work [10] such as the even-n parity problem, the multiplexer, symbolic regression and artificial ant. However, this point of view, while useful for practical benchmarking purposes, lacks generality since results are problem-dependent and it is difficult to infer more general issues relating to intrinsic GP difficulty by just looking at statistics derived from these problems.

There have been few attempts to date to characterize GP difficulty by means of a single measure. One early approach was proposed by Koza [10] and consists in calculating, for a given problem, the number of individuals that must be processed in order for a solution to be found with a given probability p (usually $p = 99\%$). This gives a number characterizing the required computational effort but it cannot be relied upon for distinguishing easy from hard in GP.

Another potentially more fruitful approach, has been to transfer to GP some of the considerations that have proven useful for studying GA difficulty. Thus, *synthetic* or *constructive* problems inspired from GA work have been devised in order to probe GP difficulty. One early example is the work of Kinnear [9] in which GP difficulty was related to the shape of the fitness landscape and analysed through the use of *correlation length*, a measure first proposed by Weinberger [16] and later used in genetic algorithms work by Manderick *et al.* [11]. Kinnear's results for GP, however, were difficult to interpret: essentially no simple relationship was found between correlation length values and GP hardness. In the same vein, Punch and coworkers [13, 14] proposed a new synthetic benchmark problem of tunable difficulty, the *Royal Tree problem*, which was inspired by the well-known *Royal Road problem* used in GA theory [12]. However, Royal Trees were used to test the effectiveness of multipopulation as compared to standard single population GP and not, to our knowledge, to gauge intrinsic GP diffi-

culty. Even though we know that there are counterexamples to the use of fitness distance correlation (*fdc*) as a tool for problem difficulty in GA [1], [15], we are also stimulated by the success of *fdc* on a large number of GA functions (see for example [3]) and by the lack of *fdc* studies in GP.

We should also mention a more recent attempt to quantify GP problem difficulty by Daida *et al.* [4]. Their approach is based on the exhaustive study of the dynamics of a single problem of the symbolic regression type, the binomial-3 problem. The binomial-3 problem is tunable thanks to the existence of a range of ephemeral random constants. This approach is interesting but it is different from ours. Whereas they try to fully understand the whole range of behaviors of this particular problem depending on the parameter interval chosen, our goal is to characterize the GP difficulty of whole classes of functions. In our opinion, the two approaches are obviously compatible and both should be pursued.

This paper is structured as follows. In section 2 we define a language that will be used at first for constructing restricted genotypes, in section 3 we define different distances between these genotypes, in section 4 we compare these distances for a particular set of functions (the trap functions) and in section 5 we use one of these distances to evaluate the problem difficulty. In section 6, we generalize the results to genotypes without syntactical restrictions. In section 7 we extend the study to Royal Trees and in section 8 we give our conclusions.

2 A RESTRICTED LANGUAGE

We have decided to study the dynamics of GP by defining a syntax for the individuals and by artificially assigning a conventional fitness value to each point in the resulting fitness landscape. For simplicity, our first step is based on an extremely simple and constrained GP model. However, we will remove this assumption in section 6 where unrestricted GP trees are used. We decided to use function and terminal sets inspired by those proposed by Punch *et al.* [13]. Thus, we will consider a set of functions A, B, C, etc. with increasing arity (an A function has arity 1, a B function has arity 2, and so on) and a single terminal X as follows: $\mathcal{F} = \{A, B, C, D, ...\}$, $\mathcal{T} = \{X\}$.

Moreover, we observe that for GAs genomes have a finite and fixed size, so the number of possible individuals in the search space is finite and numerable. For GP, this is not the case, which makes the dynamics of GP more difficult to study. In order to avoid this kind of problem, we have decided, as a first step, to limit the structures of the possible individuals of our search space. In particular, we have forbidden individuals to have as a child in the tree structure a node representing a function with an arity greater or equal to the arity of the function represented by the father. See figure 1 for some examples of legal and illegal trees.

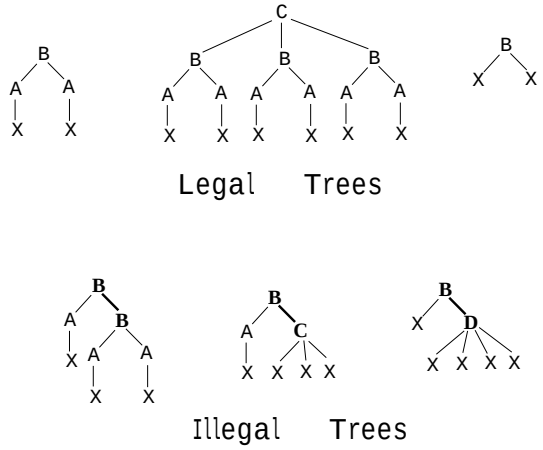


Figure 1: The three trees in the upper part are legal because every node has as son a node with a smaller arity. The other trees are illegal.

This syntactic restriction allows us to automatically limit the depth of the trees. On the other hand, this limitation obliges us to define a crossover operator such that no illegal tree is generated. This definition obviously limits what GP can do, but it is useful as a first approximation. From now on, we will refer to GP with the restrictions mentioned above as *limited GP* and to GP without the syntactical constraints as *standard GP*.

In our problems, we will consider only one individual in the search space to be the global optimum. This tree will be the perfectly balanced tree having as root the node with the maximum arity between the allowed nodes and having the maximum depth. The first tree in the upper left part of figure 1 shows the optimal tree for the set of functions $\{A, B\}$ while the second tree is the optimal one if we also insert node C in the function set. Note that this tree has C as root and three optimum trees of root B as subtrees. By the same argument, we can deduce the form of the optimum trees of any other root.

3 DISTANCE DEFINITIONS

Defining a distance between genotypes in GP is much more difficult than for GAs, given the tree structure of genotypes. For such a definition of distance for tree

structures, see for instance [8]. In the following, we will give some new definitions of distance for our restricted language, starting from a preliminary *pseudo-distance*. In the next sections, we will compare these distances and we will choose one with the aim of measuring the difficulty of problems for GP.

3.1 THE FIRST STEP: d_1

Let T_1 and T_2 be two trees. Then, we define

$$d_1(T_1, T_2) = |weight(T_1) - weight(T_2)|$$

where:

$\forall T\ weight(T) = 1 \cdot n_X(T) + 2 \cdot n_A(T) +$

$3 \cdot n_B(T) + 4 \cdot n_C(T) + ...$

and: $n_X(T)$ is the number of symbols X in the tree T, $n_A(T)$ is the number of symbols A in the tree T, and so on.

3.1.1 Problems of d_1

Figure 2 shows a set of trees with their weights, defined as in 3.1.

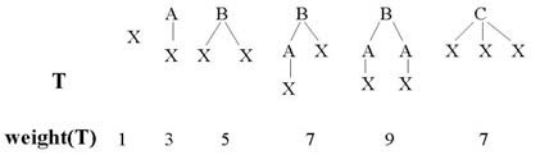


Figure 2: Some examples of legal trees with their weights, according to the definition of paragraph 3.1.

From figure 2, we can see that two trees can have a distance = 0 between each other, without being the same individual, and also having quite different structures, as is the case for trees number four and six, counting from the left in figure 2. For this reason, we called d_1 a *pseudo-distance*. The next definition, in a sense, extends d_1.

3.2 d_2: SYMBOLS WITH ROOT PRIZES

To overcome the problem presented in 3.1, we define a new concept of distance as follows:

- Each tree with root i must have a greater weight than all the trees with root j, if and only if $j < i$, where:
  - $i, j \in \{X, A, B, C, D, ...\}$
  - we consider an order such that: $X < A < B < C < D < ...$
- Since the simpler trees have a difference = 2 in the weights between each other, we give a prize to each root, such that the lighter tree of root $succ(i)$ has the weight of the heaviest tree of root i plus 2, where we impose that: $..., C = succ(B), B = succ(A), A = succ(X)$

Example

Let T_1 be the fifth tree of figure 2 and T_2 be the sixth. T_1, which is the heaviest tree of root B has a weight = 9. If we don't give any prize to the root, T_2 has root = 7. Since we want T_2 to have a weight = 11, we have to give a prize of 4 to all the trees with root C. Iterating this reasoning, we get that the prize for trees with root D must be = 28, the prize for trees with root E must be = 148, the prize for trees with root F must be = 788, the prize for trees with root G must be = 4688, and so on. With this new definition of distance, all the different trees have different weights, and so they have positive distances between each other.

Thus, the formal definition of d_2 can be given as follows: Let T_1 and T_2 be two trees. Then, we define

$$d_2(T_1, T_2) = |weight(T_1) - weight(T_2)|$$

where:

$\forall T\ weight(T) = 1 \cdot n_X(T) + 2 \cdot n_A(T) +$

$3 \cdot n_B(T) + 4 \cdot n_C(T) + ... + prize(T)$

Even though d_2 is more similar than d_1 to our idea of distance, it has a problem too: two trees that are symmetrical about the vertical have the same weights and thus the distance between them is 0, even though they are not the same tree. This problem is overcome by d_3 defined in the next section.

3.3 d_3: RECURSIVE DISTANCE

This distance has been defined with the idea of extending it to the case of general trees (i.e. trees without the limitations exposed in section 2). According to this definition, the distance of a tree T_1 from a tree T_2 can be calculated by the following formula:

$$d_3(T_1, T_2, k) = \begin{cases} k + |td(T_1) - td(T_2)| \\ \quad \text{if } root(T_1) \neq root(T_2) \\ 0 \\ \quad \text{if } td(T_1) = td(T_2) = 0 \\ \sum_{i=1}^{n(T_1)} \frac{d_3(s_i(T_1), s_i(T_2), k-1)}{n(T_1)} \\ \quad \text{otherwise} \end{cases}$$

where:

- $root(T)$ is a function that returns the root of the tree T.
- $td(T)$ is a function that returns the depth of the tree T, where the depth of the tree containing only the node X is considered to be $= 0$.
- $n(T)$ is a function that returns the number of sons of the root of the tree T (i.e. if the root of T is A it returns 1, if it is B it returns 2 and so on).
- $s_i(T)$ is the i^{th} subtree of the root of tree T.
- k is a constant that we use in order for two trees with different roots but the same depth to have a positive distance. It is useful to have smaller values of k for the nested recursive cases, so we insert k between the parameters of function d, and we pass $k-1$ to the recursive calls. The choice of the value of k is arbitrary, with the only restriction that it has to never become negative with the recursive calls, so it must be at least as large as the maximum depth allowed for the trees.

In the above formula, the upper limit of the sum is equal to the number of sons of the root of the tree T_1. We note that this quantity can be substituted by the number of sons of the root of T_2 without changing the result, since roots T_1 and T_2 have the same number of son nodes. In fact, recursion is applied only when T_1 and T_2 have the same root.

4 COMPARISON OF DISTANCES WITH TRAP FUNCTIONS

A good way to test the distances is using the *trap functions* [5]. Trap functions allow to define the fitness of the individuals as a function of their distance from the optimum, and the difficulty of trap functions can be changed simply by modifying two parameters. A function $f : distance \rightarrow fitness$ is a trap function if it is defined in the following way:

$$f(d) = \begin{cases} 1 - \frac{d}{B} & \text{if } d \leq B \\ \frac{R \cdot (d - B)}{1 - B} & \text{elsewhere} \end{cases}$$

where d is the distance of the current individual from the global optimum, and B and R are constants $\in [0, 1]$. B allows to set the width of the attractive basin for each optimum and R sets their relative importance. Figure 3 depicts a trap function with $B = 0.2$ and $R = 0.8$, where we also see that there is a second local optimum at a maximum distance from the global one. The difficulty of trap functions decreases as the value

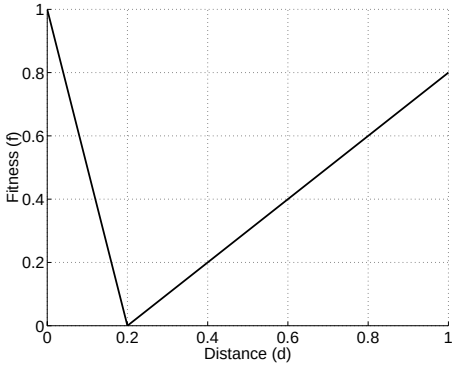


Figure 3: The graphic of a trap function with $B = 0.2$ and $R = 0.8$. Note that distances and fitnesses are normalized into the interval $[0, 1]$.

of B increases, while it increases as the value of R decreases. By keeping R constant and changing B, we are able to define a set S of trap functions of different difficulties. What we expect from a "good" distance is that GP is able to find the global optimum for the so-defined easy trap functions and not for the difficult ones. To measure the success rate to the global optimum for a function, we define a measure called *performance* (p), defined as the number of executions for which the global optimum has been found in less than 200 generations, divided by the total number of executions (100 in our experiments). Then, we perform a series of experiments on the set S and we choose the distance for which we get the best performance for the trap-easy functions to be used in the remaining of this work. All our experiments have been done with the following parameters: population size = 200, tournament selection with size = 10, crossover probability = 95%, mutation probability = 0%, maximum node allowed = F, and two different curves have been drawn for each

series of experiments, respectively with $R = 0.8$ and $R = 0.2$. Results are shown in figures 4 and 5. In

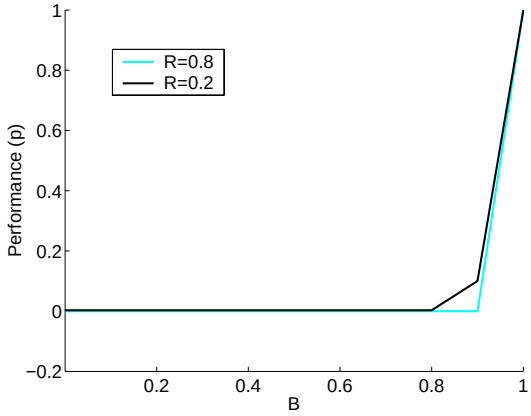


Figure 4: Curve of the performance as a function of the constant B, with $R = 0.2$ and $R = 0.8$, for distance 2.

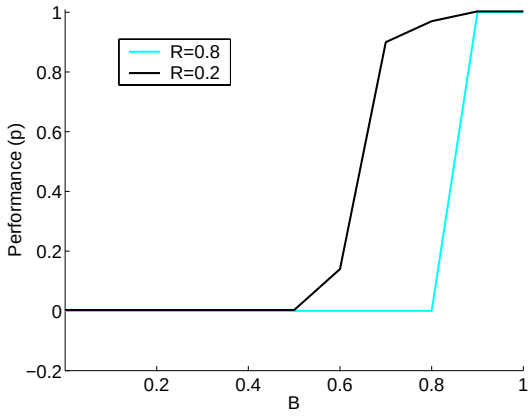


Figure 5: Curve of the performance as a function of the constant B, with $R = 0.2$ and $R = 0.8$, for distance 3.

figure 4 we can see that for d_2 we have always a performance $= 0$ except for the function with $R = 0.2$ and $B = 0.9$, for the function with $R = 0.2$ and $B = 1$, and for the function with $R = 0.8$ and $B = 1$. Of these three functions, the two with $B = 1$ reach a performance of 1. Figure 5 shows the same kind of results for d_3, for which a larger number of functions have a positive performance. Moreover, even if this result is not shown here, we observe that solutions, when using d_3 as distance, are found in a lower number of generations than with d_2. As a further confirmation of our results, in table 1, we report some other performance results calculated on another set of trap functions. From the table and from the figures it appears that d_3 is the one that shows the best performances for the highest

p	dist 1	dist 2	dist 3
B = 0.2, R =0.8	0	0	0
B = 0.3,R = 0.7	0	0	0
B = 0.4, R = 0.6	0	0	0
B = 0.5, R = 0.5	0	0	0
B = 0.6, R = 0.4	0	0	0
B = 0.7, R = 0.3	0	0	0.64
B = 0.8, R = 0.2	0	0	0.97
B = 0.9, R = 0.1	0	0.9	1
B = 1.0, R = 0.0	1	1	1

Table 1: Some values of the performance (p) for some trap functions calculated with the distances defined in the text.

number of easy trap functions, and therefore it will be used it in the following.

5 FITNESS DISTANCE CORRELATION

We now describe a heuristic that should allow us to measure the difficulty of problems to be solved with GP. An approach that has been proposed [7] states that what makes a problem hard for GAs is the relationship between fitness and the distance of the genotypes from the optimum. The easiest way to measure the extent to which the fitness function values are correlated with distance to a global optimum is to examine a problem with known optima, take *a sample* of individuals and compute the *fitness distance correlation* (*fdc*), given the set of (fitness, distance) pairs. By the way, correlation is only one of the possible ways of studying the relationship between fitness and distance. In some cases, *fdc* reveals itself to be a poor summary statistic. In such situations, examining the scatter plots of fitness versus distance to the optimum is more useful. Some works on GAs confirm previous results about the importance of a good *fdc*, and exemplify the existence of non-artificial problem, such as TSP, exhibiting this property. According to Altenberg [1], the fact that the *fdc* is only a statistical and static measure, based on a distance which is apparently only bound to mutation (like Hamming distance in GAs), implies two assumptions: either Hamming distance is connected to the way genetic algorithms work, or this relation exists in a fortuitous way among the test set chosen by Jones [7, 6]. In fact, counterexamples have been found for which this relation does not hold, and which, therefore, deceive the *fdc*. Since there seems to be no relation between re-combination operators and Hamming distance, and that mutation is supposed to play a marginal role in GAs, Altenberg claims that it is possible to construct a counterexample. The counterexample he constructs is GA-easy, but

the correlation between distance and fitness to optimum is null by construction. Furthermore, the observation of the scatter plot gives no more information. This counterexample deceives Jones' conjecture which claims that if the *fdc* is close to 0 and if the scatter plot exhibits no particular structure, then the problem is GA-difficult. Moreover, Quick *et al.* [15] construct a class of problems, called ridge functions, which are GA-easy with a high positive correlation. While the Altenberg's counterexample is prone to discussion, in particular on the definition of the GA-easiness, the counterexample of Quick *et al.* is clear: there are functions that the *fdc* predicts misleading and which are in fact easy. Nevertheless, these two counterexamples exploit known weaknesses of the *fdc*: its nullity for the symmetrical functions and the low contribution of a particular path in the global calculation. Besides, Quick *et al.* recognize that the *fdc* calculated with the points actually sampled by the GA gives better results. Nevertheless, the success of the *fdc* on a large number of GA functions remains an unsolved question. Collard *et al.* [2] bring some elements of response, exhibiting a correlation between Hamming distance and instability implied by crossover.

Formally, given a set $F = \{f_1, f_2, ..., f_n\}$ of n individual fitnesses and a corresponding set $D = \{d_1, d_2, ..., d_n\}$ of the n distances to the nearest global optimum, *fdc* is defined as: $fdc = \frac{C_{FD}}{\sigma_F \sigma_D}$, where

$$C_{FD} = \frac{1}{n} \sum_{i=1}^{n} (f_i - \overline{f})(d_i - \overline{d})$$

is the covariance of F and D and σ_F, σ_D, $\overline{f}$ and $\overline{d}$ are the standard deviations and means of F and D. As we hope that fitness increases as distance to a global optimum decreases, we expect that with an ideal fitness function *fdc* will assume the value of -1. According to Jones [7], GA problems can be classified in three classes , depending on the value of the coefficient *fdc*:

- **Misleading** ($fdc \geq 0.15$), in which fitness increases with distance.
- **Difficult** ($-0.15 < fdc < 0.15$) in which there is virtually no correlation between fitness and distance.
- **Straightforward** ($fdc \leq -0.15$) in which fitness increases as the global optimum approaches.

The second class corresponds to problems for which the difficulty can't be estimated, because the coefficient *fdc* doesn't bring any information.

Our goal is to check if the same properties are also valid for GP on trap functions using distance d_3. For this reason, we have calculated the performance p and the coefficient *fdc* for various trap functions changing the values of the constants B and R. In these experiments, *fdc* has been computed via a sample of 4000 randomly chosen individuals. Figures 6 and 7 show the results of these experiments in a tridimensional space.

These results show that for GP and with distance

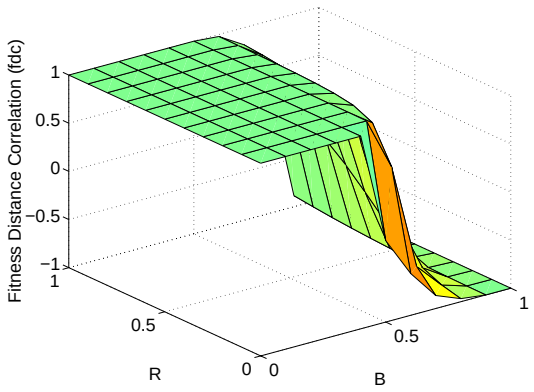


Figure 6: The values of the correlation *fdc* for the trap functions.

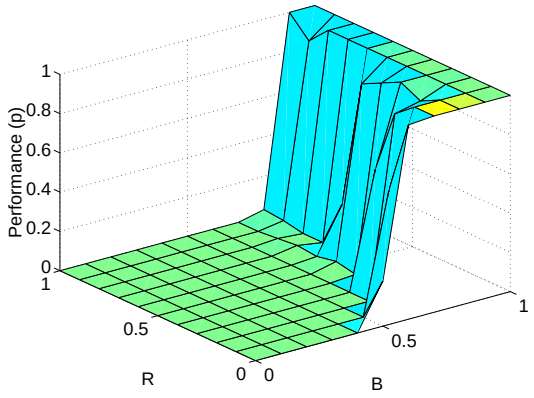


Figure 7: The values of the performance p for the trap functions.

d_3, we obtain approximately the same ranges as for GAs with unitation. In particular, from this figures, we can see that the performance is around 1 when the correlation is around -1, the performance is around 0 when the correlation is around 1 and the performance is comprised between 0 and 1 when the correlation is comprised between -1 and 1. These are exactly the results we were expecting.

6 EXTENSION TO STANDARD GP

The constraints we have imposed until now allow the creation and the survival of a restricted set of individuals (see section 2). Now we want to release this restriction, with the only obvious limitation of a maximum depth constraint. Thus, we consider the same sets of functions $\mathcal{F} = \{A, B, C, D, ...\}$ and terminals $\mathcal{T} = \{X\}$ as before but we allow the creation of any buildable tree with these symbols and with a maximum depth of 17. As before, we impose only one tree to be the optimum and, as a first step, we consider the same tree that was used for limited GP (see 6.1).

As a measure of the distance between trees, we have decided to use distance d_3 for two main reasons: it works with standard GP, and it is the distance that gives the highest performances with limited GP. Some experiments not reported here have shown that, if we consider GP without syntactical limitations, and we consider F as the function with the maximum arity, the convergence for trap functions is rather slow. In fact, no global optimum for a trap function was ever found before generation 600. This behavior is no doubt due to the huge increase in the search space when going from limited to standard GP and is also confirmed in [13]. For this reason, we have decided to perform our studies eliminating F from the function set, only after having observed that this doesn't change the main results.

6.1 FDC RESULTS

We have calculated p and fdc (see section 5) for various trap functions for the same optimum considered in limited GP. Figures 8 and 9 show the results of these experiments. From these figures we can see that the same fdc ranges also hold for standard GP with distance d_3 (see section 5) . The results are essentially the same as in the case of limited GP, with the only difference that in the case of standard GP, the number of trap functions for which $-0.15 < fdc < 0.15$ is considerably smaller. Moreover, in the case of standard GP, the number of trap functions for which $fdc = -1$ is larger than in the case of limited GP. This is partly due to the fact that we have considered F as the maximum arity function for limited GP and E for standard GP. But some results not reported here show that this would also be the case if we considered F for standard GP. Thus, for standard GP, there is a larger number of trap functions for which we can predict the difficulty, while the number of functions for which the difficulty is unpredictable is considerably smaller. In any case, the most important result is that, even for standard GP with distance d_3, the fitness distance correlation is a good measure for predicting the difficulty of problems for the class of trap functions.

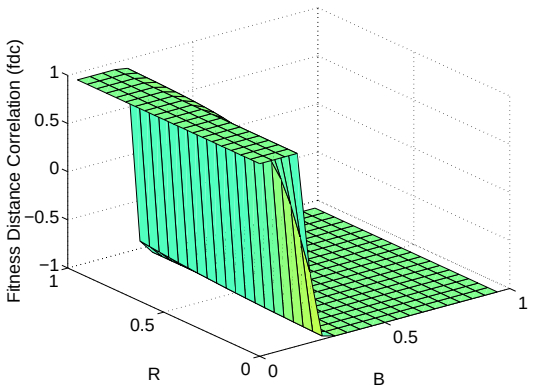


Figure 8: The values of the correlation fdc in the case of standard GP for some trap functions obtained by changing the values of the constants B and R.

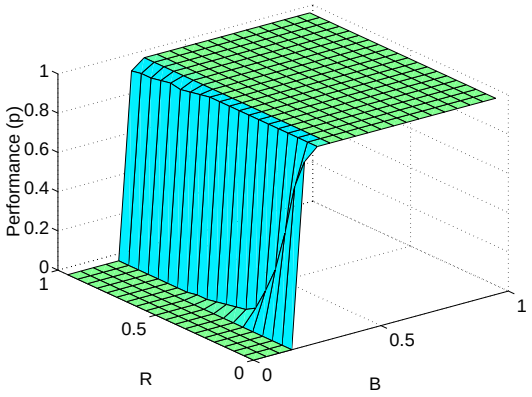


Figure 9: The values of the performance p in the case of standard GP for some trap functions obtained by changing the values of the constants B and R.

6.2 RESULTS WITH A DIFFERENT OPTIMUM

In order to understand whether the results are dependent on the particular optimum chosen, we perform the same experiments with the tree shown in figure 10 as the optimum (see 6.2). This tree has an irregular structure and it is different from the optimum used until now. Values of p and fdc for various trap functions are shown in figures 11 and 12, where it appears that, even if the choice of the optimum has an influence on difficulty, the same fdc ranges previously found (see section 5) hold, and fdc with distance d_3 is confirmed to be a good measure for predicting problems difficulty. In the next section, we study the behaviour

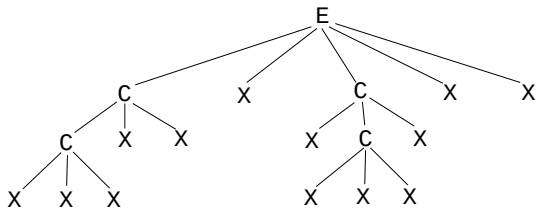


Figure 10: The tree used as optimum in the experiments of section 6.2.

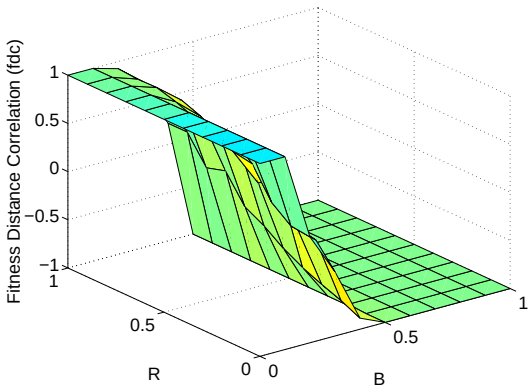


Figure 11: The values of the correlation *fdc* in the case of standard GP for some trap functions obtained by changing the values of the constants B and R. The optimum tree is the one shown in figure 10.

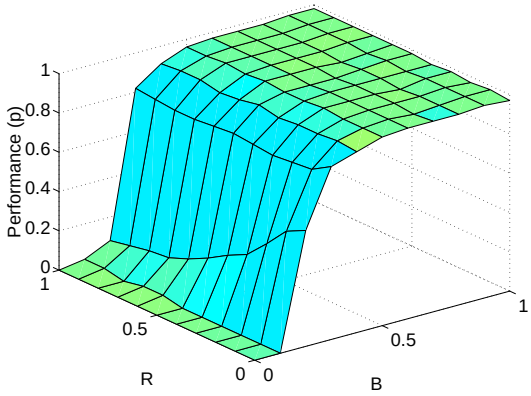


Figure 12: The values of the performance p in the case of standard GP for some trap functions obtained by changing the values of the constants B and R. The optimum tree is the one shown in figure 10.

of *fdc* for another class of functions.

7 ROYAL TREES

The last functions we take into account in this paper are the Royal Trees proposed by Punch and coworkers [13, 14]. These functions are based on the same language that was used in section 6, but the fitness is not calculated on the basis of the distance from the optimum (as it was the case for the trap functions); instead, the following algorithm is used: the raw fitness of a tree (or any subtree) is the score of its root. Each function calculates its score by summing the weighted scores of its direct children. If the child is a perfect tree of the appropriate level (for instance, a complete level-C tree beneath a D node), then the score of that subtree, times a *FullBonus* weight, is added to the score of the root. If the child has a correct root but is not a perfect tree, then the weight is *PartialBonus*. If the child's root is incorrect, then the weight is *Penalty*. After scoring the root, if the function is itself the root of a perfect tree, the final sum is multiplied by *CompleteBonus*. Values used here are: *FullBonus* = 2, *PartialBonus* = 1, *Penalty* = 0.0001, *CompleteBonus* = 2. Results on the study of *fdc* are shown in table 2. In this table, p (respectively p_1, p_2, p_3) indicates the number of executions for which the global optimum has been found in less than 200 (respectively 300, 400, 500) generations, divided by the total number of executions (100 in our experiments). From the table we can see that *fdc* correctly predicts the difficulty of level-A, level-B, level-C, and level-D functions. Level-E function is predicted by the *fdc* to be "straightforward" and it actually is, if we consider that the global optimum is found with a rate of 79% before generation 500. Level-F function is predicted to be "difficult" (where difficult means that the *fdc* doesn't give information on function hardness), and the global optimum is never found before generation 500. Finally, the level-G tree is predicted to be "misleading" (in accord with Punch [13, 14]). In conclusion, it appears that Royal Trees are a synthetic GP problem that effectively spans the classes of difficulty as described by the *fdc*.

8 CONCLUSIONS AND FUTURE WORK

In this work, we have shown that, at least for Trap Functions and Royal Trees, fitness distance correlation is a reasonable way of quantifying GP difficulty. In view of some counterexamples that have been mentioned in the text, it remains to be seen whether this measure extends to other cases such as typical GP benchmarks. This work is only a first step towards the characterization of GP difficulty from a fitness landscape point of view. In the future, we plan to extend our research to other classes of functions, to other distances, for instance the one defined in [8], and to fitness landscapes induced by operators other than standard

Root	*fdc*	*fdc* prevision	p	p_1	p_2	p_3
B	-0.45	straightf.	1	1	1	1
C	-0.33	straightf.	1	1	1	1
D	-0.26	straightf.	0.77	0.81	0.81	0.81
E	-0.22	straightf.	0.42	0.62	0.74	0.79
F	0.035	difficult	0	0	0	0
G	0.26	misleading	0	0	0	0

Table 2: Results of *fdc* for the Royal Trees

crossover, especially various kinds of tree mutations, as mutation seems to play an important role when using *fdc*. We also plan to look for a better measure than performance to identify the success rate of functions, possibly independent from the maximum number of generations chosen.

References

[1] L. Altenberg. Fitness distance correlation analysis: an instructive counterexemple. In T. Back, editor, *Seventh International Conference on Genetic Algorithms*, pages 57–64. Morgan Kaufmann, 1997.

[2] M. Clergue and P. Collard. Genetic heuristic for search space exploration. In *International Joint Conference on Artificial Intelligence (iJCAI'99)*, pages 1218–1224. Ed. Morgan Kaufmann, 1999.

[3] P. Collard, M. Clergue, and F. Bonnin. Misleading functions designed from alternation. In *Congress on Evolutionary Computation (CEC'2000)*, pages 1056–1063. IEEE Press, Piscataway, NJ, 2000.

[4] J. M. Daida, R. Bertram, S. Stanhope, J. Khoo, S. Chaudhary, and O. Chaudhary. What makes a problem GP-hard? analysis of a tunably difficult problem in genetic programming. *Genetic Programming and Evolvable Machines*, 2:165–191, 2001.

[5] K. Deb and D. E. Goldberg. Analyzing deception in trap functions. In D. Whitley, editor, *Foundations of Genetic Algorithms, 2*, pages 93–108. Morgan Kaufmann, 1993.

[6] T. Jones. *Evolutionary Algorithms, Fitness Landscapes and Search*. PhD thesis, University of New Mexico, Albuquerque, 1995.

[7] T. Jones and S. Forrest. Fitness distance correlation as a measure of problem difficulty for genetic algorithms. In L. J. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 184–192. Morgan Kaufmann, 1995.

[8] R. Keller and W. Banzhaf. Explicit maintenance of genotypic diversity on genospaces. Unpublished, 1994. http://ls11-www.informatik.uni-dortmund.de/people/banzhaf/gp.html.

[9] K. E. Kinnear. Fitness landscapes and difficulty in genetic programming. In *Proceedings of the First IEEEConference on Evolutionary Computing*, pages 142–147. IEEE Press, Piscataway, NY, 1994.

[10] J. R. Koza. *Genetic Programming*. The MIT Press, Cambridge, Massachusetts, 1992.

[11] B. Manderick, M. de Weger, and P. Spiessens. The genetic algorithm and the structure of the fitness landscape. In R. K. Belew and L. B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 143–150. Morgan Kaufmann, 1991.

[12] M. Mitchell, S. Forrest, and J. Holland. The royal road for genetic algorithms: fitness landscapes and ga performance. In F. J. Varela and P. Bourgine, editors, *Toward a Practice of Autonomous Systems, Proceedings of the First European Conference on Artificial Life*, pages 245–254. The MIT Press, 1992.

[13] B. Punch, D. Zongker, and E. Goodman. The royal tree problem, a benchmark for single and multiple population genetic programming. In P. Angeline and K. Kinnear, editors, *Advances in Genetic Programming 2*, pages 299–316, Cambridge, MA, 1996. The MIT Press.

[14] W. Punch. How effective are multiple populations in genetic programming. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, D. Goldberg, H. Iba, and R. L. Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 308–313, San Francisco, CA, 1998. Morgan Kaufmann.

[15] R.J. Quick, V.J. Rayward-Smith, and G.D. Smith. Fitness distance correlation and ridge functions. In *Fifth Conference on Parallel Problems Solving from Nature (PPSN'98)*, pages 77–86. Springer-Verlag, Heidelberg, 1998.

[16] E. D. Weinberger. Correlated and uncorrelated fitness landscapes and how to tell the difference. *Biol. Cybern.*, 63:325–336, 1990.

Size Control via Size Fair Genetic Operators in the PushGP Genetic Programming System

Raphael Crawford-Marks
Box 820
Hampshire College
Amherst, MA 01002
rpc01@hampshire.edu

Lee Spector
Cognitive Science
Hampshire College
Amherst, MA 01002
lspector@hampshire.edu

Abstract

The growth of program size during evolution (code "bloat") is a well-documented and well-studied problem in genetic programming. This paper examines the use of "size fair" genetic operators to combat code bloat in the PushGP genetic programming system. Size fair operators are compared to naive operators and to operators that use "node selection" as described by Koza. The effects of the operator choices are assessed in runs on symbolic regression, parity and multiplexor problems (2,700 runs in total). The results show that the size fair operators control bloat well while producing unusually parsimonious solutions. The computational effort required to find a solution using size fair operators is about equal to, or slightly better than, the effort required using the comparison operators.

1 INTRODUCTION

Code bloat in genetic programming has been documented since the field came into existence a decade ago. In the past few years bloat has been studied extensively, with researchers examining the causes of bloat as well as testing new operators designed to limit code bloat (D'haeseleer, 1994; Angeline, 1994; Langdon and Poli, 1997; Poli and Langdon, 1997; Banzhaf, et al., 1998; Soule and Foster, 1998; Langdon, et al., 1999; Francone, et al., 1999; Langdon, 1999; Luke, 2000). Recently, "size fair" operators have been shown to limit bloat significantly without decreasing the problem-solving ability of the genetic programming system (Langdon, et al., 1999; Langdon, 1999).

This paper extends Langdon's work by testing size fair operators in a genetic programming system that uses unusual representations for programs. The PushGP system is conventional in most respects but it manipulates and produces Push programs rather than the Lisp-like program trees used in more conventional genetic programming systems (for example (Koza, 1992)). Push programs, like Lisp programs, are variably sized strings of symbols and balanced (possibly nested) sets of parentheses. On the other hand, Push programs are interpreted quite differently from Lisp programs; Push program interpretation is more similar to the interpretation of stack-based languages like Forth or Postscript. The applicability of Langdon's work to PushGP is therefore an interesting test of the generality of his findings.

A detailed description of the Push programming language is beyond the scope of this paper; see (Spector and Robinson, 2002) for a full introduction and language reference, or (Spector, 2001; Robinson, 2001) for brief introductions. The essential feature of the Push language for the present study is just that the programs are syntactically similar to, yet semantically quite different from, the Lisp-like programs used in traditional genetic programming systems. Push's unique structure supports many enhancements to genetic programming systems (for example, efficient and fully automatic evolution of modular programs) but none of these are relevant to the present study; see (Spector and Robinson, 2002) for details.

In Langdon's prior work he tested a 50%-150% fair mutation operator in stochastic problem solving systems (e.g., hill climbing and simulated annealing systems) but not specifically in genetic programming systems. In this study we apply a variant of this operator in PushGP and demonstrate its utility for genetic programming. We also describe a new size fair crossover operator and describe the performance of the size fair operators in all possible combinations with

naive operators and operators that use node selection (a technique based on Koza's 90%-10% tree/leaf selection method (Koza, 1992)).

2 Bloat in PushGP

Nested parentheses in the Push syntax make it possible to model a Push program as a tree, and therfore to apply standard, tree-based genetic operators in PushGP. The original version of PushGP used what we will call "naive" operators which were meant to be as simple as possible while capturing the essential ideas of traditional genetic programming operators. The naive mutation operator selects a random "point" of the program to replace, with each point having an equal probability of being chosen. Each symbol and each parenthesized expression in the program counts as a point. The chosen point is then replaced with a new randomly generated expression, which will have a size uniformly selected from the range $[1, n]$, where n is a system parameter. The naive crossover operator selects random points in both parent programs (again with all points having an equal probability of being chosen) and returns a copy of one of the parents with the chosen point from its mate replacing its own chosen point.[1]

Bloat is quite strong in PushGP when the naive operators are used, in large part because the naive mutation operator generates random subtrees that are larger, on average, than the subtrees they replace. In order to keep program lengths manageable, PushGP implements a size ceiling. Any program exceeding the size ceiling is discarded, and a clone of one of its parents is used in its place; in the tables below we refer to this as a "size limit replication." Most runs with the naive crossover and mutation operators exhibit rapid code bloat, with program sizes climbing steadily toward the size ceiling. It has been shown that when put to work on simple symbolic regression problems, 20%-45% of the children were over the size limit at Generation 50, and thus discarded in favor of a clone of the parent (Robinson, 2001). (Robinson, 2001) also discovered that the naive crossover and mutation operators were more likely to select leaf nodes than internal nodes, resulting in little variation in the internal structure of programs across the population.

3 What Causes Bloat?

Code bloat is not a phenomenon particular to GP. It has been shown to occur in several non-GP stochastic search techniques (Langdon, 1998). There are many studies on the origins of bloat. Some findings suggest that because there are more large programs than small ones in a search space, fitness-based selection on average finds larger programs with better fitness than smaller or equal-sized programs (Langdon and Poli, 1997). Others suggest that bloat occurs as an evolutionary defense mechanism against destructive operators. Traditional crossover and mutation can often be fatal when applied to a small, fit program as they randomly rip out a chunk of the fit program and replace it with a different random chunk of program. Thus, programs evolve "introns" (segments of neutral code) as a means to preserve fitness when subjected to destructive evolutionary operators (Nordin and Banzhaf, 1995). Similar to defense theory, (Soule and Foster, 1998) suggest that individuals are penalized when a large chunk of code is removed, but not so when a large chunk is inserted, thus driving up the size of the program. This is called "removal bias". More recently, (Luke, 2000) suggested that introns are not the cause of code growth, but rather a symptom, and that a bias towards deeper crossover points drives code growth.

The underlying cause of bloat is still open to debate. What is universally agreed upon, however, is that bloat occurs and often has detrimental effects on the improvement in fitness in genetic programming runs. In addition, it clearly slows down genetic programming runs by consuming CPU cycles and large amounts of memory.

4 New Operators

We studied four variations of the genetic operators (two variations of mutation, two of crossover), in addition to the naive operators described above.

Node Selection, a method described in (Koza, 1992), chooses an internal node 90% of the time and a leaf node 10% of the time for either mutation or crossover. Node selection was implemented both for mutation and crossover in PushGP.

"Size Fair" crossover and mutation operators are operators that on average produce children of the same size as their parents. The size fair mutation operator we use is identical to the 50%-150% operator described in (Langdon, 1998; Langdon, et al. 1999), except that it produces mutations of length $\ell \pm \frac{\ell}{4}$ instead of $\ell \pm \frac{\ell}{2}$, where ℓ is the number of points in the subtree to be mutated.[2] The size distribution of the replacement

[1] The random code generator is described in (Spector and Robinson, 2002).

[2] The fraction $\frac{\ell}{4}$ was chosen arbitrarily, prior to reading Langdon's work. We assume the specific fraction has little effect on performance.

subtrees (and thus the resulting children) is uniform.

Our new crossover operator, Fair Crossover, differs from the size fair crossover operator described in (Langdon, 1999). Langdon's operator selects the first crossover point at random from Parent 1. The size (ℓ) of the subtree at the first crossover point is calculated, and the lengths of all subtrees in Parent 2 are also calculated. All subtrees from Parent 2 whose size is larger than $1+2\ell$ are excluded. This limits the amount by which the child can increase in size to $1+\ell$ larger than its parent. For the remaining subtrees, the number that are smaller, the same size and larger than ℓ are each counted, along with the mean size difference for the larger and smaller subtrees. A roulette wheel is used to select the size of the subtree to be crossed over. The selection method is biased using calculated mean size differences such that on average there is no change in program size after crossover is performed.

With our new Fair Crossover operator, the first crossover point is selected at random from Parent 1, and the length of the subtree at that point is measured. Then a randomly selected subtree from Parent 2 is measured. If its length is within the range $\ell \pm \frac{\ell}{4}$ (where ℓ is the length of the subtree from the first parent), the subtree from Parent 2 replaces the subtree in Parent 1. If not, another subtree is randomly selected from Parent 2, and the test is repeated. If no subtrees are found within the range $\ell \pm \frac{\ell}{4}$ after n attempts, the subtree whose size was closest to $\ell \pm \frac{\ell}{4}$ is used in crossover. The size distribution of replacement subtrees is dependent on the parents, and may not be uniform.[3]

For the experiments described in this paper, Fair Crossover would perform 20 retries before giving up and using the subtree with the closest length. We will call this a "punt". In the 300 runs on symbolic regression of a sextic polynomial that used Fair Crossover, the operator punted just over 80% of the time. This means that if 2250 crossovers were performed, the operator only found replacement subtrees within the length $\ell \pm \frac{\ell}{4}$ about 450 times. However, since the subtree whose size is *closest* to $\ell \pm \frac{\ell}{4}$ is used after 20 tries, Fair Crossover still has an effect close to that of a size fair operator. In the same 300 runs, replacement subtrees found by Fair Crossover were on average only 0.8 points larger than the original subtree. Given the low bloat observed when Fair Crossover is used, it appears that while Fair Crossover may not be perfectly size fair, it is quite close.

[3]Fair Crossover was used instead of Langdon's size fair crossover operator because it was simpler to implement.

```
- push-base-type:
      dup, pop, swap, rep, =, set, get, convert,
      pull, pulldup, noop
  - number: +, -, *, /, >, <
    - integer: pull, pulldup, /
  - boolean: not, and, or
  - expression:
      quote, car, cdr, cons, list, append, subst,
      container, length, size, atom, null, nth,
      nthcdr, member, position, contains, insert,
      extract, instructions, replace-atoms,
      discrepancy
    - code: do, do*, if, map
```

Figure 1: Push function set used for the PushGP runs.

5 Results

All combinations of crossover and mutation operators were used in sets of 100 independent genetic programming runs on 3 different problems: Sextic Regression, Even-5 Parity, and 6-Bit Multiplexor.

For all problems, the population size was 5000, the program size ceiling 50 points, and the runs were limited to 50 generations. The size of mutant subtrees added by the Naive Mutation operator was limited to 10 points. The operator rates were 45% crossover, 45% mutation and 10% straight reproduction. The tournament size was 7. The function set is listed in Figure 1. See (Spector and Robinson, 2002) additional information on the Push functions.

Computational Effort was computed in the standard way, as described by Koza on pages 99 through 103 of (Koza, 1994). To summarize briefly, one conducts a large number of runs with the same parameters (except random seeds) and begins by calculating $P(M, i)$, the cumulative probability of success by generation i using a population of size M. For each generation i this is simply the total number of runs that succeeded on or before the ith generation, divided by the total number of runs conducted. From $P(M, i)$ one can calculate $I(M, i, z)$, the number of individuals that must be processed to produce a solution by generation i with probability greater than z. Following the convention in the literature we use a value of z=99%. $I(M, i, z)$ can be calculated using the following formula:

$$I(M, i, z) = M * (i + 1) * \left\lceil \frac{\log(1 - z)}{\log(1 - P(M, i))} \right\rceil$$

The more steeply the graph of $I(M, i, z)$ falls, and the lower its minimum, the better the genetic programming system is performing. Koza defines the minimum of $I(M, i, z)$ as the "computational effort" re-

Table 1: Results for symbolic regression of $x^6 - 2x^4 + x^2$, sorted by computational effort.

Crossover Method	Mutation Method	Successful Runs	Average Solution Size	Average Size Limit Replications (Gen. 25)	Average Size Limit Replications (Gen. 49)	Computational Effort
Fair	Node Sel	93/100	31.29	363.00	715.71	450000
Fair	Naive	85/100	32.66	965.76	1456.47	480000
Node Sel	Fair	87/100	39.29	725.46	948.71	495000
Naive	Node Sel	88/100	37.10	941.20	1216.17	540000
Fair	Fair	88/100	21.77	10.19	64.85	540000
Naive	Fair	87/100	33.63	362.70	587.21	585000
Naive	Naive	71/100	38.17	1519.24	1920.52	800000
Node Sel	Node Sel	76/100	40.58	1328.78	1576.00	820000
Node Sel	Naive	61/100	42.36	2024.82	2280.97	960000

Table 2: Results for Even-5 Parity, sorted by computational effort.

Crossover Method	Mutation Method	Successful Runs	Average Solution Size	Average Size Limit Replications (Gen. 25)	Average Size Limit Replications (Gen. 49)	Computational Effort
Fair	Naive	100/100	34.85	318.50	*	240000
Naive	Naive	98/100	36.62	1201.22	850.00	250000
Fair	Node Sel	99/100	29.32	29.69	281.00	270000
Naive	Node Sel	99/100	36.06	413.65	386.00	280000
Node Sel	Naive	100/100	41.58	1659.48	*	290000
Naive	Fair	96/100	29.56	214.06	258.60	310000
Node Sel	Fair	96/100	31.75	342.15	793.50	320000
Fair	Fair	97/100	20.99	8.23	0.00	330000
Node Sel	Node Sel	98/100	37.89	657.51	1016.00	350000

* All runs completed before 49th Generation

Table 3: Results for 6-Bit Multiplexor, sorted by computational effort.

Crossover Method	Mutation Method	Successful Runs	Average Solution Size	Average Size Limit Replications (Gen. 25)	Average Size Limit Replications (Gen. 49)	Computational Effort
Fair	Fair	30/100	19.80	0.46	28.56	1870000
Fair	Node Sel	36/100	27.58	71.41	428.67	1885000
Naive	Fair	32/100	27.53	127.00	410.82	2080000
Naive	Node Sel	26/100	30.96	389.41	749.47	2520000
Fair	Naive	26/100	32.27	623.75	1388.20	2635000
Node Sel	Naive	23/100	37.57	1375.40	1725.29	2835000
Node Sel	Fair	26/100	27.96	325.13	673.92	3120000
Naive	Naive	26/100	37.92	972.08	1519.34	3200000
Node Sel	Node Sel	18/100	31.11	697.06	1014.76	4320000

quired to solve the problem. Computational effort is not a perfect measure (see, for example, (Luke and Panait, 2002)) but we believe it is sufficient for the modest uses to which it is put here.

5.1 Symbolic Regression of $x^6 - 2x^4 + x^2$

As shown in Table 1, the combination of Fair Crossover and Node Selection Mutation yielded the most solutions, least computational effort and the second-most parsimonious solution sizes. The combination of Fair Mutation and Fair Crossover yielded the second most solutions (tied with Naive Crossover and Node Selection mutation) and the most parsimonious ones as well (by nearly 10 points), but scored in the middle of the field in terms of computational effort. Notable also is that the operators causing the greatest amount of bloat (and thus the greatest number of replications due to hitting the size ceiling) finished in the last three spots in terms of solutions found and computational effort.

5.2 Even-5 Parity

Even-5 Parity is a fairly easy problem for PushGP to solve, as shown by the high number of solutions found. Interestingly, one of the least successful combinations from the regression runs, Naive Crossover with Naive Mutation, scored just behind Fair Crossover with Naive Mutation in terms of computational effort. Again, Fair Crossover with Fair Mutation found the most parsimonious solutions by nearly 10 points, and kept replications down to almost nothing, but scored next to last in terms of computational effort.

5.3 6-Bit Multiplexor

When applied to the 6-bit Multiplexor problem different operators performed best. The pairing of size fair mutation and crossover found the third most solutions with the least computational effort. The size fair operators also found the most parsimonious solutions, beating the next best pair of operators by almost 8 points. The rest of the top performers all had performed well in previous runs. Performing particularly poorly was the pairing of Node Selection mutation and crossover, which found the fewest solutions and required the most computational effort.

6 Discussion

The efficacy of the size fair operators in controlling bloat and in producing parsimonious solutions is clear from the data. Certainly for the cases in which fair mutation and fair crossover were used together the improvements in these measures were dramatic. Additionally, in many cases the use of just one size fair operator, in conjunction with a non-size-fair operator, seems to confer advantages.

It should come as no surprise that it is impossible to declare one operator or combination of operators as clearly being better than the rest with respect to the computational effort required to find a solution. However, we do note that all of the runs with size fair operators performed at least reasonably well; the use of size fair operators does not appear to be detrimental with respect to this measure. We also note that the better combinations often included one size fair operator and one non-size-fair operator. One could speculate that size fair operators, used by themselves, slow the genetic programming system in its progress to larger areas of the search space (where solutions are more plentiful) thus increasing the time it takes to find solutions. If so then one might further speculate that the judicious mixing of non-size-fair operators, which can have more dramatic impacts on program size, with size fair operators would be the best way to encourage robust problem solving performance. More research would be required to confirm or falsify these speculations.

7 Conclusions

The size fair operators examined in this work appear to control bloat well and to encourage the production of parsimonious solutions without negative impacts on the computational effort required to find a solution. This is important because unchecked bloat limits the applicability of genetic programming by requiring exorbitant computational resources, and because naive approaches to bloat control can change the system's evolutionary dynamics in ways that make it harder to find solutions. Solution parsimony is also important because it simplifies the work of humans who must interpret the output of genetic programming systems, and because more parsimonious solutions may in some cases also be more general.

This work was conducted using the PushGP system which is similar to traditional genetic programming systems in some ways but different from them in others. The reported work extends Langdon's earlier work, demonstrating that the idea of size fair operators has utility across a broader range of program representations. The obvious next step is to repeat this study, using Langdon's operators and the new size fair crossover operator that we have developed, in a more traditional genetic programming system. If they per-

form as well in such a follow-up study, controlling bloat and producing parsimonious solutions without sacrificing the problem-solving capacity of the system, then we would recommend their wide-spread adoption.

Acknowledgments

Thanks are due to Benjamin Lefstein for observations and suggestions, and to Alan Robinson for keeping the cluster computer running.

This effort was sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30502-00-2-0611. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, or the U.S. Government.

This research was also made possible by generous funding from Hampshire College to the Institute for Computational Intelligence at Hampshire College.

References

Angeline, P. 1994. Genetic Programming and Emergent Intelligence. In *Advances in Genetic Programming*. MIT Press, pp. 75-98.

Banzhaf, W., P. Nordin, R. E. Keller and F. D. Francone. 1998. Genetic Programming: An Introduction. Morgan Kaufmann Publishers.

D'haeseleer, P. 1994. Context Preserving Crossover in Genetic Programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*. IEEE Press, pp. 256-261.

Francone, F. D., M. Conrads, W. Banzhaf, and P. Nordin. 1999. Homologous Crossover in Genetic Programming. In *Proceedings of the Genetic and Evolutionary Computation Conference GECCO-99*. Morgan Kaufmann, pp. 1021 - 1026.

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

Koza, J. R. 1994. *Genetic Programming II: Automatic Discovery of Reusable Programs*, MIT Press, 1994.

Langdon, W. B. 1999. Size Fair and Homologous Tree Genetic Programming Crossovers. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-99*. Morgan Kaufmann, pp. 1092 - 1097.

Langdon, W. B., T. Soule, R. Poli, and J. A. Foster. 1999. The evolution of size and shape. In *Advances in Genetic Programming 3*. MIT Press, Chapter 8, pages 163 - 190.

Langdon, W. B. 1998. The Evolution of Size in Variable Length Representations. In *1998 IEEE International Conference on Evolutionary Computation*. IEEE Press, pp. 633 - 638.

Langdon, W. B. and R. Poli. 1997. Fitness Causes Bloat. In *Second On-line World Conference on Soft Computing in Engineering Design and Manufacturing*. Springer-Verlag London.

Luke, S. 2000. Code Growth Is Not Caused By Introns. In *Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference*. IEEE Press, pp. 228-235.

Luke, S., and L. Panait. 2002. Is the Perfect the Enemy of the Good? In Langdon, W. B., et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2002*. San Francisco, CA: Morgan Kaufmann Publishers.

Nordin, P. and W. Banzhaf. 1995. Complexity compression and evolution. In *Genetic Algorithms: Proceedings of the Sixth International Conference*. Morgan Kaufmann Publishers.

Poli, R. and W. B. Langdon. 1997. Genetic Programming with One-Point Crossover. In *Soft Computing in Engineering Design and Manufacturing*. Springer-Verlag London, pp. 180-189.

Robinson, A. 2001. "Genetic Programming: Theory, Implementation, and the Evolution of Unconstrained Solutions," Hampshire College Division III (senior) thesis.
http://hampshire.edu/lspector/robinson-div3.pdf.

Soule, T. and J. A. Foster. 1998. Removal bias: a new cause of code growth in tree based evolutionary programming. In *1998 IEEE International Conference on Evolutionary Computation*. IEEE Press, pp. 781-186.

Spector, L. 2001. Autoconstructive Evolution: Push, PushGP, and Pushpop. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2001*. Morgan Kaufmann Publishers.

Spector, L., and A. Robinson. 2002. Genetic Programming and Autoconstructive Evolution with the Push

Programming Language. In *Genetic Programming and Evolvable Machines*. Vol. 3, No. 1, pp. 7–40.

Evolving chess playing programs

R. Groß, K. Albrecht, W. Kantschik and W. Banzhaf
Dept. of Computer Science
University of Dortmund
Dortmund, Germany

Abstract

This contribution introduces a hybrid GP/ES system for the evolution of chess playing computer programs. We discuss the basic system and examine its performance in comparison to pre-existing algorithms of the type alpha-beta and its improved variants. We can show that evolution is able to outperform these algorithms both in terms of efficiency and strength.

1 Introduction

Computer programs capable of playing the game of chess have been designed for more than 40 years, starting with the first working program that was reported in 1958 [BR58]. Since then countless numbers of programs were developed and appropriate hardware was designed.

This article introduces a chess program which learns to play the game of chess under limited resources. We want to demonstrate the capabilities of Computational Intelligence (CI) methods to improve the abilities of known algorithms. More precisely we investigate the power of Genetic Programming (GP) [BNKF98] and Evolutionary Strategies (ES) [Sch96] using the example of computer chess. The relevance of computer chess is probably comparable to that of fruit flies in genetics, it is a laboratory system with paradigmatic character.

In previous work we have studied the evolution of chess-playing skills from scratch [BKA$^+$00]. In this contribution we do not want to create an entirely new program to play chess. Instead, we start with a scaffolding algorithm which can perform the task already and use a hybrid of GP and ES to find new and better heuristics for this algorithm. We try to improve a simple algorithm, the *alpha-beta* algorithm. In order to evolve good standard heuristics we use evolutionary techniques.

It is very time consuming to evolve chess playing individuals. Thus the basis of our evolutionary system is a distributed computing environment on the internet called ***qoopy***. Distributed computing is necessary because of the high costs of one fitness evaluation for a chess program. Each individual has to perform several games against computer programs and a game might last several hours in worst case. The additional computer power needed to perform the evolution of chess programs is borrowed from participating users worldwide through the internet.

The performance of our evolved programs is neither comparable to Deep Blue [Kor97] nor to other computer chess programs playing at expert level. This was not intended at the present stage of development.

2 The Chess Individuals

We use an *alpha-beta* algorithm [Sch89] as the kernel of an individual which is enhanced by GP- and ES-modules. The goal of these modules is the evolution of smart strategies for the middle game. So no opening books or end game databases have been used to integrate knowledge for situations where a tree search exhibits weak performance. Also, GP/ES individuals always play white and standard black players are employed to evaluate the individuals. The black players are fixed chess programs which think a certain number of moves ahead and then choose the best move (according to the minimax-principle, restricted by the search horizon) [Bea99, BK77]. The individuals are limited to search not more than an average of 100,000 nodes per move to ensure an acceptable execution speed.

Like standard chess programs, individuals perform a tree search [IHU95]. In particular, they use an alpha-beta-algorithm. Three parts of this algorithm are

Table 1: The pseudocode shows the $\alpha\beta$ algorithm with the evolutionary parts (bold).

```
αβ_max (position K; integer α, β) {
  integer i, j, value ;
  nodes = nodes + 1;
  IF POSITION_DECIDED(K) THEN
   RETURN position module (K);
  IF (depth == maxdepth) THEN
   RETURN position module (K);
  restdepth = depth module (restdepth);
  IF (((restdepth == 0) OR
   (nodes > maxnodes))
   AND (depth >= mindepth)) THEN
    RETURN position module (K);
  determine successor positions K.1, ..., K.w;
  move ordering module (K.1, ..., K.w);
  value = α;
  FOR j = 1 TO w DO {
    restdepthBackup = restdepth;
    restdepth = restdepth − 1;
    value=max(value, αβ_min(K.j, value, β));
    restdepth = restdepthBackup;
    IF value ≥ β THEN {
      ADJUST KILLERTABLE;
      RETURN value;
     }
  }
  RETURN value;
}
```

evolved:

- The depth module, which determines the remaining search depth for the given node.
- The move ordering module, which changes the ordering of all possible moves.
- The position module, which returns a value for a given chess position.

Evaluation of a chess individual is performed in the following way (see table 1): The depth module determines the remaining depth for the current level in the search tree. If the position is a leaf then the position module is called to calculate a value for it. Otherwise the node (move) will be expanded and all possible moves will be calculated. Subsequently the move ordering module for these moves is called and changes the order of the moves, so that moves which are more important can be evaluated first in the search tree.

2.1 Depth module

Table 2: The terminal set of the depth module with a short description. With chess-specific operations the depth module receives chess knowledge.

terminal	description of the terminal
accumulator	Default register for all functions, initialized each node.
level register	Special register which holds information of the current level in the search tree, initialized each search.
search/game register	Special register, initialized each search/game.
search depth	Returns the current search depth of the tree.
search horizon	Value of the current search horizon.
piece value	Value of a piece given by the accumulator.
captured piece	Value of a captured piece, if the last move was a capture move.
alpha/beta bound	Value of alpha/beta bound.
move number	Number of current move, given by the move ordering module.
# pieces	Number of knights, bishops, rooks and pawns of the board.
# expanded nodes	Number of expanded nodes for the current position, in percent.
value of move	Value of the move which led to the current position, given by the move ordering module.
branching factor	Number of moves of the precedent position.
position score	Value of the current position, given from the position module.

The depth module decides for each position whether the search tree should be expanded and to what depth.

Normally chess programs have a fixed search horizon [PSPDB96]. This means, that after a certain number of plies the expansion in the search tree will be terminated. In contrast, the depth module should give the individual a higher flexibility in the search to avoid the search horizon effect.

The depth module has two limitations, the search depth and the amount of nodes used in a search tree. The maximal depth is 10 plies but if all moves until ply 10 would be executed approximately 10^{14} nodes would be expanded. Therefore the amount of ex-

panded nodes in the search tree was limited to 100,000 nodes on average per move. On average means, that the individual might save nodes in particular periods of the game to spend them later.

The depth module is a GP-program of branched operation sequences. The structure is an acyclic graph where each node holds a linear program and each if-then-else condition makes a decision which node of the program will be executed next (see section 2.4). Its function set holds very simple functions but its terminal set is more chess-specific, see tables 2 and 3.

Table 3: The function set of the depth module with a short description. No function of the set has chess knowledge.

function	description of the function
+, -, *, /	Arithmetic functions; result is written to the accumulator.
inc/decHorizon	Function to increment/decrement the search depth by one, but the search depth can only be increased/decreased by 2 per level.
sine	The sine function.
sigmoid	The sigmoid function $\frac{1}{1+e^{-terminal}}$.
store in level/game/ search register	Stores the terminal in the level/game/search register.
load	Loads the terminal to the accumulator.
if	If the condition is true the left branch will be executed, otherwise the right one. A condition can be a comparison of two terminals.

The depth module does not define the depth of search directly, rather it modifies how much depth is left for searching - the *restdepth*. It can be incremented or decremented by the operation *incHorizon* and *decHorizon*, or stay untouched. The *restdepth* is initialized with a value of 2. Once a move is executed the *restdepth* is automatically decremented by 1.

2.2 Position module

The position module of an individual calculates a value for the current position.

The position module is a fixed algorithm which evaluates the position by using evolved values for the different chess pieces and some structural values of a check. These values are accumulated whereas *bonus* values are added and *punish* values are subtracted - a higher value corresponds to a better position. We used a simple ES to evolve these values. The idea of this module is to find a better set of parameters than a fixed position evaluation algorithm would provide. Values of hand-written programs are determined through experience of the programmer or by taking parameters from the literature.

Our ES evolves the following weights for the position evaluation algorithm. The first two numbers in brackets reflects the range within which the values can be chosen, the last number is the standard value which was chosen for the black players (see 3.3).

- Values of different pieces: pawn [85-115, 100], knight [290-360, 340], bishop [300-370, 340], rook [440-540, 500], queen [800-1000, 900].
- Bishops in the initial position are punished [0-30, 15].
- Center pawn bonus: Pawns in the center of the chessboard get a bonus [0-30, 15].
- Doubled pawn punishment: If two pawns of the same color are at the same column [0-30, 15].
- Passed pawn bonus: A pawn having no opponent pawn on his and the neighboring columns [0-40, 20].
- Backward pawn punishment: A backward pawn has no pawn on the neighboring columns which is nearer to the first rank [0-30, 15].
- If a pawn in end game is near the first rank of the opponent it gets a promotion bonus depending on the distance, this value fixes the maximal bonus [100-500, 300].
- Two bishops bonus: If a player has both bishops it gets a bonus [0-40, 20].
- A knight gets a mobility bonus, if it can reach more than 6 fields on the board [0-30, 15].
- Knight bonus in closed position: A closed position is defined if more than 6 pawns occupy the center of the board. The center consists of the 16 fields in the center of the board [0-40, 20].
- Knight punishment: If opponent pawns are on each side in end game [0-50, 25].
- Rook bonus for a half open line: A half open line is a line with no friendly pawn that does have an enemy pawn [0-30, 15].

- Rook bonus for an open line: An open line is a line without a pawn on this line [0-30, 15].
- Rook bonus for other positional advantages [0-20, 10].
- Rook bonus: If a rook is on the same line as a passed pawn [0-30, 15].
- King punishment, if the king leaves the first rank during the opening and the middle game [0-20, 10].
- Castling bonus, if castling was done [0-40, 20].
- Castling punishment for each weakness of pawn structure (exception: end game) [0-30, 15].
- Castling punishment, if the possibility was missed [0-50, 25].
- Random value, this is a random value which will be added or subtract from the position value [0-30, 20].

The structure of the position evaluation algorithm for the chess individual and the black player is identical. However there is a difference: Values for individuals are evolved, values for black players are predefined and fixed.

2.3 Move ordering module

The move ordering module of an individual orders the moves for each chess position by assigning a real number to every possible move. The value of a move is the sum of several weighted features of the move. Moves are sorted according to these values and moves will be expanded by this order. By default the value of a feature is in $[0, 100]$.

An ES evolves the following weights for the sorting algorithm:

- Piece values in the opening/middle and end game: Each piece are assigned three values which reflect how important this piece is in the opening/middle and end game.
- Most valuable victim/Least valuable aggressor: The ratio of aggressor and victim move values is calculated. A position with a high ratio is better than one with a smaller value.
- Check: If the move leads to a check position then the move fulfilling this feature gets a bonus.
- Capture move: These are moves which can capture a piece of the opponent.
- Pawn moves that can attack a piece of the opponent.
- Pawn moves that do not attack a piece of the opponent.
- Pawn moves that lead to a promotion of a queen.
- Center activity: Pieces which move from and/or to the center of the chess board gets a bonus.
- Killer moves: Killer moves are moves which often lead to a cut in the search tree. The table contains at most 4 killer moves for each level in search tree. The table will be filled during the search, and if a move is in the killer table it gets a bonus depending on its rank, a lower and upper bound is given by this feature. Besides, the composition of the killer table which changes during search is influenced by the move ordering module of an individual.

Based on these weights, the value for moves will be calculated. Sorting of the moves is very important for the *alpha-beta* search algorithm. If the best move is visited first, the following moves don't need to be considered. A very good move ordering module results in a better performance of the *alpha-beta* algorithm.

2.4 GP structure of the depth module

The depth module of an individual, as illustrated in Figure 1, is represented by a program with nested if-then-else statements [KB01]. This representation has been developed with the goal of giving a GP-program greater flexibility to follow different execution paths for different inputs. It also achieves a reuse of the evolved code more frequently than is the case in linear GP [Nor94, NB95].

A program consists of a linear sequence of statements and if-then-else statements, that contain a bifurcation into two sequences of statements. Nested if-then-else statements are allowed up to a constant recursion depth. The resulting structure is a graph where each node contains a linear GP-program and a decision part. During the execution of the program only one path of the graph will be executed for each input.

Crossover of two programs can be realized in different ways. We have chosen the following two types. The first crossover operator selects a sequence of statements in each program. In case of selected if-then-else statements, the associated statements of the then- and

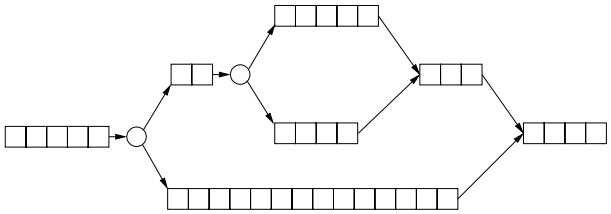


Figure 1: The representation of the GP program, which is a graph structure. The squares represents the linear programs and the circles represents the if-then-else decisions.

else- parts are selected, too. Then the selected sequences are swapped. Secondly, a swapping of branch-free sequences is allowed for an exchange of information between individuals.

Mutation is performed subsequently to crossover or independently from it. There are two types of mutation operators. The first one performs a crossover with a randomly generated individual. The second one selects a sequence of statements (in case of if-then-else statements including the statements belonging to the associated then- and else- parts). Afterwards each statement other than an if-then-else one will be mutated with an adjustable probability (see [KB02]).

3 Evolution

Evolution is based on populations of several individuals. Each individual has to be evaluated by determining its fitness. In our system a fitness case is a chess game and an individual has to play several chess games before its fitness can be assigned. We used the approach of distributed computing [GCK94] to allow for enough computing power. We developed the ***qoopy*** system in order to spread our task among the internet. As opponents of GP-/ES-programs, chess programs with fixed depth were used. Fitness was calculated based on the number of wins, losses and draws against these opponents.

In the following sections we describe this system in more detail.

3.1 Internet-wide evolution based on *qoopy*

qoopy is an environment for *distributed computing* tasks [GCK94, Far98]. It is possible to develop distributed programs for the ***qoopy*** environment and use ***qoopy*** to run these programs.

The first application of ***qoopy*** is **EvoChess**, a distributed software system which creates new chess programs in an evolutionary process. After ***qoopy*** is installed on a machine each participant runs a deme containing a variable number of individuals (default value is 20). In each deme evolution begins and, during the evolution, individuals might be copied between demes (pollination) to create offsprings. ***qoopy*** provides the necessary infrastructure for communication between demes and the connection to an application server.

The application server is necessary because ***qoopy*** has to register users being online to let them connect to each other and to exchange data. The server holds results of the internet evolution, and each deme sends its best individuals and other statistics back to the server on a regular basis.

3.2 Fitness evaluation

The fitness of an individual is a real number between 1 and 15, with higher values corresponding to better individuals. In order to determine fitness, individuals have to play chess games against fixed algorithms of strength 2, 4, 6, 8, 10, 12 and 14. For fitness evaluation an individual always plays white (see 3.3).

The result of a game is a real number between -1 and 1. It is 1 in case that the individual wins the game, -1 if the standard algorithm wins the game and 0 in case of draw. Sometimes it is obvious that one side can win or that the game has to end draw. In such a case the game is stopped to save time. In rare cases lengthy games are aborted because nothing happens anymore.[1] Then the position is evaluated and the result reflects the advantage of white (positive) or black (negative) as a value in the range of $[-1, \ldots, 1]$.

Fitness is initialized with a value of 1. Resulting values are weighted with the number of games played relative to the strength of the opponent. If, e.g., the individual loses twice and wins once against an opponent of strength 6 $(-1, -1, 1)$, this results in values $(5.0, 5.0, 7.0)$, and the fitness is 5.667.

In general, the fitness of an individual is calculated by the following function:

$$fitness = \sum_{j \in \mathcal{C}} \sum_{i=1}^{n^j} \frac{j + \text{result}_i^j}{n^j * |\mathcal{C}|}$$

Classes $\mathcal{C}$ are the classes with wins, draws and losses of the individual. These classes lie in an interval whose

[1] There are several criteria to prevent games to be canceled in *interesting* situations, e.g. when a king has been checked or a piece has been captured within the last moves.

bounds are defined by the following rules: If an individual wins all its games up to class i, these results are ignored and if an individual loses all its games from class i to the highest class, these results are ignored.

For example if an individual i wins all games of classes 2, 4, and 8, and has wins, draws and losses in the classes 6 and 10, and loses all games in the classes 12 and 14, $\mathcal{C}_i$ holds the classes 6, 8 and 10. The first rule defines the lower bound of the interval (4), and the second rule defines the upper bound of the interval (12). The first rule does not hold for class 8 because in class 6 the individual has had a draw or loss.

In general, the fitness of an individual is calculated in four phases. Thus weak individuals can be dropped from fitness evaluation in the first or second phase. Fitness evaluation in phase three and four is very CPU time expensive and we try to reduce the computation time by removing inviable programs.

In phase one the individual plays two games against 2. If the individual is very weak it can be identified by the fitness function and replaced immediately. In the second phase the individual plays against 4, 6, 8 and 10. If the fitness is at least 4.5 at the end of phase two, the evaluation is continued in phase three. In phase three the individual plays 1-2 games against 2, 4, 6, 8 and 10. Successful individuals might skip games against 2 and 4. These are individuals which win each game up to strength 6 and receive good results in games against 8 and 10. In phase four games are performed against 12 and 14. Only the best individuals play in this phase, however. Games against the standard algorithm of class 12 and 14 are very expensive.

To play more than twice against class 12 (or 14) it is required to win in one of the two games before. Every draw results in 1 point, every loss in 2 points. If the individual has more than 6 (5) points it does not play any more against class 12 (14). If the individual is good enough it will play 4 times against 12 and then 3 times against 14.

3.3 Opponents of the individuals (black players)

The opponents of individuals are chess programs which can fully traverse the search tree up to a fixed depth. We use these players to calculate the fitness of an individual, by playing against fixed programs of different search depth. Fixed programs can play to a depth of 1, 2, 3, 4, 5, 6 and 7. Each of these programs defines a corresponding fitness class of 2, 4, 6, 8, 10, 12 and 14. The value for a fitness class is the search depth multiplied with 2, so that an individual which defeats an individual of class 4 but loses against an individual of class 6 can be inserted into class 5.

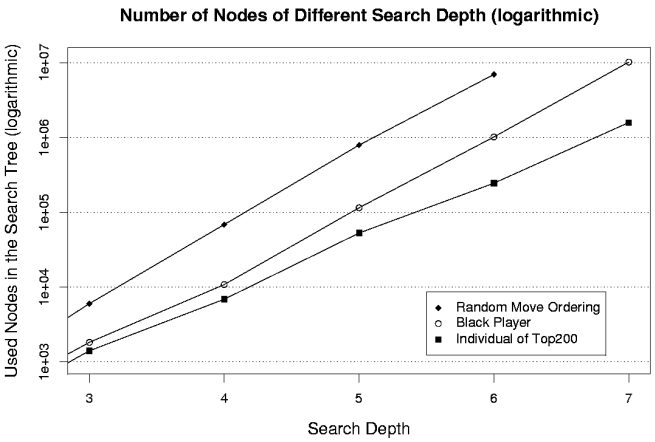


Figure 2: Average number of nodes used in the search tree for different fixed search depths. For search depth 6 the plot shows a large difference between a random move ordering and an evolved move ordering module. Even the f-negascout search algorithm requires more resources than an evolved individual. The data are average values of more than 1000 moves (from reference games).

The GP/ES individuals use a position evaluation of the same structure and the same criteria - but their weights are determined by the individual's genotype.

To reduce the number of nodes of the game tree an f-negascout algorithm [Rei89] combined with iterative deepening is performed for the black players. The f-negascout algorithm is an improved variant of alpha-beta, which is the most wide-spread tree search algorithm. Iterative deepening performs a search to depth $d-1$ before searching to depth d (recursively). In addition, so-called killer moves are stored and tried first whenever possible. Killer moves are moves which result in the cut of a subtree. This means that much of the game tree can be discarded without loss of information!

4 Results

In this section we describe the current results of an ongoing evolution on the internet.

First we look at the evolved individuals and their efficiency in search. The question is: How many resources are needed by the evolved move ordering modules in case of a fixed-depth search in comparison to other move ordering strategies. Figure 2 shows the number

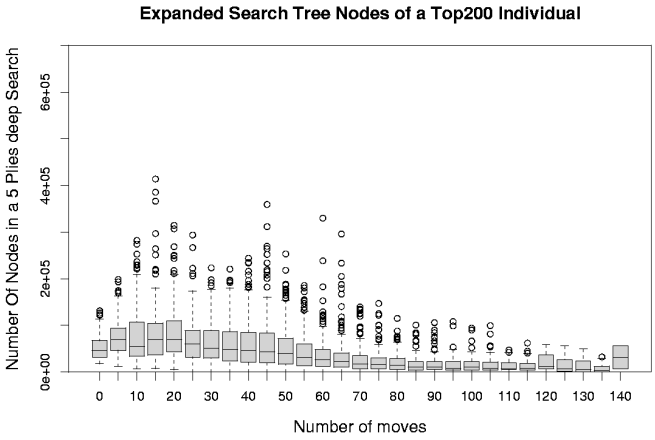


Figure 3: Box plot diagram of average number of used nodes during a game with one of an evolved individual combined with a simple fixed-depth alpha-beta algorithm, average taken over more than 5000 moves from 50 reference games. The bar in the gray boxes is the median of the data. A box represents 50 % of the data, this means that 25 % of the data lies under and over a box. The smallest and biggest *usual* values are connected with a dashed line. The circles represents outliers of the data.

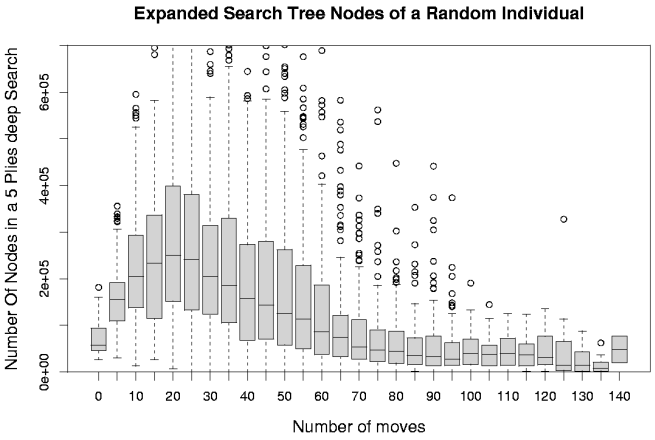


Figure 4: Box plot diagram of average number of used nodes during a game with a random individual (see also figure 3).

of nodes examined in the search tree of an alpha-beta-algorithm with a random move ordering, an evolved individual and the f-negascout algorithm (the opponent of an individual (the black player) is always an f-negascout algorithm). The figure shows that a random move ordering algorithm calculates seven million nodes with a search tree of depth 6. The f-negascout algorithm needs one million nodes. An evolved individual only needs 250,000 nodes. So evolution has managed to create individuals which perform a very efficient search through the tree.

Figures 3 and 4 show a box plot diagram investigating the number of search nodes visited by an evolved and a random move ordering module combined with a simple fixed-depth alpha-beta algorithm. The evolved individual clearly outperforms the random one. Besides, the figures show that most nodes during a game are used between ply 10 and 60.

The other aspect of the evolved chess programs is the quality of the selected moves. Currently evolution succeeded to evolve a chess playing program, with a fitness of 10.48. This means that the evolved program is better than the opponent program of fitness class 10. The fitness value was measured by a post-evaluation of best programs: An individual plays 20 games against class 8, 10 and 12, so that the fitness value is the result of 60 games. Note that the individual achieves this result by expanding on average 58,410 nodes per move in the search tree. A simple *alpha-beta* chess program needs 897,070 nodes per move for search depth of 5, which corresponds to class 10. The f-negascout algorithm which is an improved variant of *alpha-beta*, needs 120,000 nodes per move for this search depth. In other words evolution has improved the search algorithm, so that it wins by only using 50% of the resources of a f-negascout algorithm which, in turn, outperforms an *alpha-beta*-algorithm. Evolved individuals win against a simple *alpha-beta*-algorithm by using only 6% of the resources.

5 Summary and Outlook

We have shown, that it is possible to evolve chess playing individuals superior to given algorithms. At this time evolution is still going on and results are still improving.

Next we shall develop this approach by using other search algorithms as the internal structure, and by exchanging the different modules. A further feature will be that individuals will play against each other.

The ultimate goal of our approach is to beat computer programs like Deep Blue, which to this day use brute-force methods to play chess.

ACKNOWLEDGMENT

The authors gratefully acknowledge the enthusiastic support of a large group of **EvoChess** users. A

list of participants is available at http://qoopy.cs.uni-dortmund.de/qoopy_e.php?page=statistik_e . All of them have helped to produce these results and to improve both **EvoChess** and the ***qoopy*** system considerably. Support has been provided by the DFG (Deutsche Forschungsgemeinschaft), under grant Ba 1042/5-2.

SUPPLEMENTARY MATERIAL

More information on the ***qoopy*** system, the EvoChess application, and experimental data are available from:

http://www.qoopy.net.

References

[Bea99] D.F. Beal. *The Nature of Minimax Search*. PhD thesis, University of Maastricht, 1999. Diss.Nr.99-3.

[BK77] J. Birmingham and P. Kent. Tree-Searching and Tree-Pruning Techniques. In M.R.B. Clarke, editor, *Advances in Computer Chess 1*, pages 89–97. Edinburgh University Press, 1977.

[BKA+00] J. Busch, W. Kantschik, H. Aburaya, K. Albrecht, R. Gross, P. Gundlach, M. Kleefeld, A. Skusa, M. Villwock, T. Vogd, and W. Banzhaf. Evolution von GP-Agenten mit Schachwissen sowie deren Integration in ein Computerschachsystem. SYS Report SYS-01/00, ISSN 0941-4568, Systems Analysis Research Group, Univ. Dortmund, Informatik, 10 2000.

[BNKF98] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming – An Introduction On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, San Francisco und dpunkt verlag, Heidelberg, 1998.

[BR58] A. Bernstein and M. de V. Roberts. Computer v Chess-Player. *Scientific American*, 198:96–105, 1958.

[Far98] J. Farley. *Java Distributed Computing*. O'Reilly, 1998.

[GCK94] J. Dollimore G.F. Coulouris and T. Kindberg. *Distributed Systems, Concepts and Design*. Addison-Wesley, 2 edition, 1994.

[IHU95] H. Iida, K. Handa, and J. Uiterwijk. Tutoring Strategies in Game-Tree Search. *ICCA Journal*, 18(4):191–205, December 1995.

[KB01] W. Kantschik and W. Banzhaf. Linear-tree GP and its comparison with other GP structures. In J. F. Miller, M. Tomassini, P. Luca Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 302–312, Lake Como, Italy, 18-20 April 2001. Springer-Verlag.

[KB02] W. Kantschik and W. Banzhaf. Linear-graph gp - a new gp structure. In J. F. Miller, M. Tomassini, P. Luca Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2002*, LNCS. Springer-Verlag, 2002.

[Kor97] R.E. Korf. Does Deep Blue Use Artificial Intelligence? *ICCA Journal*, 20(4):243–245, December 1997.

[NB95] Peter Nordin and Wolfgang Banzhaf. Evolving turing-complete programs for a register machine with self-modifying code. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 318–325, Pittsburgh, PA, USA, 15-19 1995. Morgan Kaufmann.

[Nor94] J. P. Nordin. *A Compiling Genetic Programming System that Directly Manipulates the Machine code*, pages 311–331. MIT Press, Cambridge, 1994.

[PSPDB96] A. Plaat, J. Schaeffer, W. Pijls, and A. De Bruin. Best-first fixed-depth minimax algorithms. *Artificial Intelligence*, 87:255–293, 1996.

[Rei89] A. Reinfeld. An improvement of the scout tree search algorithm. *ICCA Journal*, 6(4):4–14, June 1989.

[Sch89] J. Schaeffer. The History Heuristic and Alpha-Beta Search Enhancements in Practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(11):1203–1212, 1989.

[Sch96] H-P. Schwefel. *Evolution and Optimum Seeking*. John Wiley & Sons, Inc., 1996.

Breeding Algebraic Structures – An Evolutionary Approach to Inductive Equational Logic Programming

Lutz Hamel
Department of Computer Science and Statistics
University of Rhode Island, Kingston, Rhode Island 02881, USA
email: hamel@cs.uri.edu

Abstract

Concept learning is the induction of a description from a set of examples. Inductive logic programming can be considered a special case of the general notion of concept learning specifically referring to the induction of first-order theories. Both concept learning and inductive logic programming can be seen as a search over all possible sentences in some representation language for sentences that correctly explain the examples and also generalize to other sentences that are part of that concept. In this paper we explore inductive logic programming with equational logic as the representation language and genetic programming as the underlying search paradigm. Equational logic is the logic of substituting equals for equals with algebras as models and term rewriting as operational semantics.

1 INTRODUCTION

The aim of concept learning is to induce a description of a concept from a set of examples. Typically the set of examples are ground sentences in a particular representation language. Concept learning can be seen as a search over all possible sentences in the representation language for sentences that correctly explain the examples and also generalize to other sentences that are part of that concept [11, 16]. Inductive logic programming (ILP) can be considered a special case of the general notion of concept learning specifically referring to the induction of first-order theories as descriptions of concepts [17].

Specialized search mechanisms for specific representation languages have been devised over the years. For example, in the propositional setting we have Quinlan's entropy based decision tree algorithm ID3 [21]. In the first-order logic setting we have Muggleton's inductive logic programming system Progol whose underlying search paradigm is based on inverting logical entailment [19].

Since concept learning and inductive logic programming imply complex searches, it is natural to ask whether evolutionary algorithms are applicable in this area. Briefly, evolutionary algorithms are a class of algorithms that traverse complex search spaces by mimicking natural selection. The algorithms maintain a large population of individuals with different characteristics where each individual represents a point in the search space. By exerting selective pressures on this population, fitter individuals representing better solution points according to the search criteria will emerge from the population. These fitter individuals in turn are allowed to reproduce in a preferential manner in subsequent generations increasing the overall fitness of the population. At the end of the run the fittest individuals in the final population represent the final solution points in a complex search space [9, 14]. To date evolutionary algorithms, particularly genetic programming systems, have successfully been applied to concept learning and inductive logic programming tasks in a variety of formalisms. For example, they have been successfully applied in the propositional case [13], in the first-order logic setting [24, 10], as well as in the higher-order functional logic programming setting [11].

In this paper we examine an evolutionary approach to concept learning based on another formalism – many-sorted first-order equational logic. Equational logic is the logic of substituting equals with equals. Here the examples are ground equations and the induced concept descriptions are first-order equational theories. We have implemented a prototype by incorporating a specialized genetic programming engine into the equational logic programming system and algebraic speci-

fication language OBJ3 [5, 6]. Informally, the system operates by maintaining a population of candidate theories that are evaluated against the examples using OBJ3's deductive machinery. The fittest theories are allowed to reproduce in accordance to standard genetic programming practices. Because of the fact that we are inducing first-order equational theories we tend to refer to this approach as *inductive equational logic programming*.

This search based view of inductive logic programming is a very operational view. It is possible to formulate a semantics to inductive logic programming that is independent of any particular search strategy. We will discuss this *normal semantics* to ILP in more detail below. Equational theories have a very strong notion of sorts and operators; i.e., they have a very strong notion of signatures. We recast the first-order logic normal semantics for ILP into an algebraic light that deals with the strong notion of signature effectively. This algebraic formulation of the normal semantics for ILP forms the basis of our system implementation with the genetic programming strategy as the operational semantics.

The system most closely related to ours is the FLIP system [2, 7]. It also concerns itself with the induction of first-order equational theories from ground equations. However, the FLIP system uses inverse narrowing as a search strategy instead of the evolutionary approach as advocated here. On a technical equational logic level the FLIP system deals with signatures only implicitly, which means that by design it is limited to single-sorted equational logic.

The rest of this paper is organized as follows. Section 2 provides a brief introduction to many-sorted equational logic, algebra and term rewriting. In Section 3 we examine the normal semantics for inductive logic programming. We develop an algebraic semantics for our setting in Section 4. Section 5 sketches our system implementation. In Section 6 we take a critical look at results obtained with the preliminary implementation so far. We end with the conclusions in Section 7.

2 EQUATIONAL LOGIC

Equational logic is the logic of substituting equals for equals with algebras as models and term rewriting as the operational semantics [15, 23, 1]. The following formalizes these notions.

An equational signature defines a set of sort symbols and a set of operator or function symbols.

Definition 1 *An* **equational signature** *is a pair* (S, Σ)*, where* S *is a set of sorts and* Σ *is an* $(S^* \times S)$*-sorted set of operation names. The operator* $\sigma \in \Sigma_{w,s}$ *is said to have arity* $w \in S^*$ *and sort* $s \in S$*. Usually we abbreviate* (S, Σ) *to* Σ*.* [1]

We define Σ-algebras as models for these signatures as follows:

Definition 2 *Given a many sorted signature* Σ*, a* Σ**-algebra** A *consists of the following:*

- *an* S*-sorted set, usually denoted* A*, called the* **carrier** *of the algebra,*
- *a* **constant** $A_\sigma \in A_s$ *for each* $s \in S$ *and* $\sigma \in \Sigma_{[],s}$*,*
- *an* **operation** $A_\sigma \colon A_w \to A_s$*, for each non-empty list* $w = s1 \ldots sn \in S^*$*, and each* $s \in S$ *and* $\sigma \in \Sigma_{w,s}$*, where* $A_w = A_{s1} \times \ldots \times A_{sn}$*.*

Mappings between signatures map sorts to sorts and operator symbols to operator symbols.

Definition 3 *An* **equational signature morphism** *is a pair of mappings* $\phi = (f, g) \colon (S, \Sigma) \to (S', \Sigma')$*, we write* $\phi \colon \Sigma \to \Sigma'$*.*

A theory is an equational signature with a collection of equations.

Definition 4 *A* Σ**-theory** *is a pair* (Σ, E) *where* Σ *is an equational signature and* E *is a set of* Σ**-equations**. *Each equation in* E *has the form* $(\forall X) l = r$*, where* X *is a set of variables distinct from the equational signature* Σ *and* $l, r \in T_\Sigma(X)$ *are terms over the set* Σ *and* X*. If* $X = \emptyset$*, that is,* l *and* r *contain no variables, then we say the equation is* **ground**. *When there is no confusion* Σ*-theories are referred to as theories and are denoted by their collection of equations, in this case* E*.*

The above can easily be extended to conditional equations[2]. However, without loss of generality we continue the discussion here based on unconditional equations only. Also, our current prototype solely

[1] Notation: Let S be a set, then S^* denotes the set of all finite lists of elements from S, including the empty list denoted by []. Given an operation f from S into a set B, $f \colon S \to B$, the operation f^* denotes the extension of f from a single input value to a list of input values, $f^* \colon S^* \to B$, and is defined as follows: $f^*(sw) = f(s) f^*(w)$ and $f^*([]) = []$, where $s \in S$ and $w \in S^*$.

[2] Consider the conditional equation, $(\forall X) l = r$ if c, which is interpreted as meaning the equality holds if the condition c is true.

considers the evolution of theories with unconditional equations.

The models of a theory are the Σ-algebras that satisfy the equations. Intuitively, an algebra satisfies an equation if and only if the left and right sides of the equation are equal under all assignments of the variables. More formally:

Definition 5 *A Σ-algebra A* **satisfies** *a Σ-equation $(\forall X)l = r$ iff $\overline{\theta}(l) = \overline{\theta}(r)$ for all assignments $\overline{\theta}\colon T_\Sigma(X) \to A$. We write $A \models e$ to indicate that A satisfies the equation e.*

We define satisfaction for theories as follows:

Definition 6 *Given a theory $T = (\Sigma, E)$, a Σ-algebra A is a T-model if A satisfies each equation $e \in E$. We write $A \models T$ or $A \models E$.*

In general there are many algebras that satisfy a particular theory. We also say that the class of algebras that satisfy a particular equational theory represent the denotational semantics of that theory.

Semantic entailment of an equation from a theory is defined as follows.

Definition 7 *An equation e is* **semantically entailed** *by a theory (Σ, E), write $E \models e$, iff $A \models E$ implies $A \models e$ for all Σ-algebras A.*

Mappings between theories are defined as theory morphisms.

Definition 8 *Given two theories $T = (\Sigma, E)$ and $T' = (\Sigma', E')$, then a* **theory morphism** *$\phi\colon T \to T'$ is a signature morphism $\phi\colon \Sigma \to \Sigma'$ such that $E' \models \phi(e)$, for all $e \in E$.*

In other words, the signature morphism ϕ is a theory morphism if the translated equations of the source theory T are semantically entailed by the target theory T'.

Goguen and Burstall have shown within the framework of institutions [1] that the following holds for many sorted algebra[3]:

Theorem 9 *Given the theories $T = (\Sigma, E)$ and $T' = (\Sigma', E')$, the theory morphism $\phi\colon T \to T'$, and the T'-algebra A', then $A' \models_{\Sigma'} \phi(e) \Rightarrow \phi A' \models_\Sigma e$, for all $e \in E$.*

[3]Actually, Goguen and Burstall have shown the much more powerful result that the implication holds as an equivalence relation. However, for our purposes here we only need the implication.

In other words, if we can show that a given model of the target theory satisfies the translated equations of the source theory, it follows that the reduct of this model, $\phi A'$, also satisfies the source theory, thus, the models behave as expected.

Our approach to equational logic so far has been purely model theoretic. A proof theory for many-sorted equational logic is defined by the following *rules of deduction*. Given a signature Σ and a set of Σ-equations, the following are the rules for deriving new equations [15] (here t, u, and v denote terms over the signature Σ and an appropriate variable set):

1. *Reflexivity.* Each equation $(\forall X)t = t$ is derivable.
2. *Symmetry.* If $(\forall X)t = t'$ is derivable, then so is $(\forall X)t' = t$.
3. *Transitivity.* If the equations $(\forall X)t = t'$, $(\forall X)t' = t''$ are derivable, then so is $(\forall X)t = t''$.
4. *Substitutivity.* If $(\forall X)t1 = t2$ of sort s is derivable, if $x \in X$ is of sort s', and if $(\forall Y)u1 = u2$ of sort s' is derivable, then so is $(\forall Z)v1 = v2$, where $Z = (X - \{x\}) \cup Y$, $vj = tj(x \leftarrow uj)$ for $j = 1, 2$, and '$tj(x \leftarrow uj)$' denotes the result of substituting uj for x in tj.
5. *Abstraction.* If $(\forall X)t = t'$ is derivable, if y is a variable of sort s and y is not in X, then $(\forall X \cup \{y\})t = t'$ is also derivable.
6. *Concretion.* Let us say that a sort s is *void* in a signature Σ iff $T_{\Sigma,s} = \emptyset$. Now, if $(\forall X)t = t'$ is derivable, if $x \in X_s$ does not appear in either t or t', and if s is non-void, then $(\forall X - \{x\})t = t'$ is also derivable.

Given a theory (Σ, E), we say that an equation $(\forall X)t = t'$ is *deducible* from E if there is a deduction from E using rules 1-6 whose last equation is $(\forall X)t = t'$ [23]. We write: $E \vdash (\forall X)t = t'$.

The model theoretic and the proof theoretic approaches to equational logic are related by the notion of soundness and completeness.

Theorem 10 (Soundness and Completeness of Equational Logic) *Given an equational theory (Σ, E), an arbitrary equation $(\forall X)t = t'$ is semantically entailed iff $(\forall X)t = t'$ is deducible from E. Formally, $E \models (\forall X)t = t'$ iff $E \vdash (\forall X)t = t'$, where $t, t' \in T_\Sigma(X)$.*

This theorem is very convenient, since it lets us use equational deduction to check the theory morphism

conditions above which plays an important part in our system implementation.

Term rewriting [12, 15] can be considered an efficient implementation of *unidirectional equational deduction* by viewing equations as rewrite rules from left to right. Given a Σ-equation $(\forall X)t = t'$, consider: a term t_0 can be rewritten into a term t_1 provided that t_0 contains a subterm that is a substitution instance of the left side t of the equation. Then t_1 is the result of replacing the substitution instance of t with the appropriate substitution instance of t' in t_0. Given this, every term can be rewritten to a unique canonical form under mild conditions on the set E of Σ-equations, such as every variable of a right side of an equation must also appear in the left side. This forms the basis of the operational semantics of the OBJ specification language [5, 6].

3 INDUCTIVE LOGIC PROGRAMMING

Traditionally, inductive logic programming has concerned itself with the induction of first-order logic theories from facts and background knowledge. The *normal* semantics for ILP is usually stated as follows [3, 20],

Definition 11 *Given a set B of horn clause definitions (background theory), a set P of ground facts to be entailed (positive examples), a set N of ground facts not to be entailed (negative examples), and a hypothesis language L, then a construct $H \in L$ is an* **hypothesis** *if*

$$B \cup H \models p, \text{ for every } p \in P \text{ (Completeness)},$$
$$B \cup H \not\models n, \text{ for every } n \in N \text{ (Consistency)}.$$

Here, L is the set of all well-formed logical formulae over a fixed vocabulary. *Completeness* states that the conjunction of the background and the hypothesis entail the positive facts. *Consistency* states that the background and the hypothesis do not entail the negative facts or counter examples. Logical entailment is derived by interpreting the clauses in the appropriate Herbrand models [22].

Please note that this semantic definition does not say anything about the quality of a particular hypothesis. In fact, it is interesting to note that this semantic definition admits a number of trivial solutions; for instance, let $H = P$. Also consider the case where $B \models p$ for every $p \in P$. Typically, the weighing of one hypothesis over another is left to the operational or search semantics of an ILP system. In practical ILP systems trivial solutions like the ones above are typically immediately dismissed by the system on its search for an "optimal" hypothesis, since these trivial solutions tend not to pass a set of performance criteria when compared to other more general hypotheses.

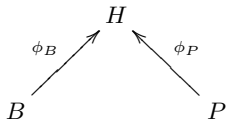
4 AN ALGEBRAIC SEMANTICS

The above semantics for ILP treats signatures implicitly. However, type information and signatures play a central role in many-sorted equational logic. Therefore, we recast the above semantics in an algebraic setting based on signatures, equational theories, and theory morphisms. We start by defining what we mean by facts.

Definition 12 *A theory (Σ, E) is called* **Σ-facts** *if each $e \in E$ is a ground equation.*

This allows us to define our notion of induced theory.

Definition 13 *Given a background theory $B = (\Sigma_B, E_B)$, positive facts $P = (\Sigma_P, E_P)$, and negative facts $N = (\Sigma_N, E_N)$, then an* **induced theory** *$H = (\Sigma_H, E_H)$, is a theory with a pair of mappings ϕ_B and ϕ_P*



such that

- $\phi_B : B \to H$ *is a theory morphism,*
- $\phi_P : P \to H$ *is a theory morphism,*
- *and $H \not\models \phi_N(e)$, for all $e \in E_N$, and signature morphism $\phi_N : \Sigma_N \to \Sigma_H$.*

Our induced theory is not unlike the hypothesis in the normal semantics. In fact, by making ϕ_B an inclusion morphism we have the algebraic equivalent formulation of the normal semantics for ILP. We like the added generality our semantics supports and will explore this in future implementations. Currently, the prototype interprets ϕ_B as the inclusion morphism.

Taking a closer look at ϕ_B, from the definition we have $\phi_B : B \to H$ is a theory morphism if $H \models \phi_B(e)$, for each $e \in E_B$. This is equivalent of saying that in order for this mapping to be valid the induced theory must semantically entail the given background knowledge. Of course this holds trivially if ϕ_B is the inclusion morphism.

A closer look at the theory morphism ϕ_P that maps the positive facts into the induced theory reveals a similar

relationship. Again from the definition, $\phi_P : P \rightarrow H$ is a theory morphism if $H \models \phi_P(e)$, for each $e \in E_P$. This can be considered the algebraic formulation of the *completeness* criteria of the normal ILP semantics. Please note, by replacing the semantic entailment with proof theoretic deduction which follows from the soundness and completeness of equational logic we obtain a computable relation. This is precisely what we use in our system implementation below.

The last part of the definition above is the algebraic formulation of the *consistency* statement: negative facts should not be entailed by the induced theory. Similar to the normal semantics our algebraic semantics says nothing about the quality of the induced theory. This is left to the search semantics of the system; in our case this is left to the genetic programming engine.

So far we have treated models that satisfy H implicitly. It is interesting to take a look at the models *per se*.

Proposition 14 *Given an induced theory H, with the background theory B, the positive facts P, and the negative facts N, then each model m that satisfies the induced theory H and is consistent with the negative facts N also satisfies the background theory B and the positive facts P.*

Proof: From the previous section we know that for every theory morphism $\phi : T \rightarrow T'$ and a model $m' \models T'$ there is a reduct $\phi m'$ such that $\phi m' \models T$. Let us assume that there exists a model m that satisfies the induced theory H and is consistent with N, i.e., $m \models H$ and $m \not\models N$. We then have two reducts along the theory morphisms $\phi_B : B \rightarrow H$ and $\phi_P : P \rightarrow H$, namely $\phi_B m$ and $\phi_P m$, respectively, where $\phi_B m \models B$ and $\phi_P m \models P$. Thus, consistent models that satisfy the induced theory H have reducts along the theory morphisms and behave as expected. □

5 SYSTEM IMPLEMENTATION

We have implemented our prototype system within the OBJ3 algebraic specification system [5, 6]. OBJ3 implements many-sorted equational logic[4] with algebras as its denotational semantics and many-sorted term rewriting as its operational semantics.

The following specification of a stack of elements can be considered a prototypical OBJ3 specification.

```
obj STACK is sorts Stack Element .
  op empty : -> Stack .
  op push  : Stack Element -> Stack .
  op top   : Stack -> Element
  op pop   : Stack -> Stack .
  var X : Element .  var S : Stack .
  eq top(push(S,X)) = X .
  eq pop(push(S,X)) = S .
endo
```

The first line of the specification names the theory and also defines two sorts; namely, `Stack` and `Element`. The following four lines define the operations on the stack. We then define the variables we need in the equations on the following two lines.

The current prototype incorporates a genetic programming engine based on Koza's canonical LISP implementation [14] into the OBJ3 system. The engine performs the following steps given a (possibly empty) background theory and the facts:

1. Compute initial (random) population of candidate theories;
2. Evaluate each candidate theory's fitness using the OBJ3 rewrite engine;
3. Perform candidate theory reproduction according to the genetic programming paradigm;
4. Compute new population of candidate theories;
5. Goto step 2 or stop if target criteria have been met.

This series of steps does not significantly differ from the standard genetic programming paradigm. The fittest individual of the final population is considered to be the induced theory satisfying the given facts.

A couple of things are noteworthy. The signatures of the candidate theories are computed using the signature morphism constructions underlying the algebraic semantics outlined above. Both, for the background theory as well as for the positive facts we let the signature morphisms be inclusions. In order to complete the candidate theories the system adds equations to the computed signatures according the to the genetic programming paradigm.

For the negative facts we take advantage of OBJ3's builtin boolean operator `=/=`. This operator allows us to recast negative facts as inequality relations that need to hold in the candidate theories. In effect, these inequalities become positive facts and we treat them as such by adding them to the positive facts theory. Consequently we set the negative fact theory to the empty theory. This technique facilitates the coding for the genetic programming engine, since the notion

[4] Actually, OBJ3 implements order-sorted equational logic, which means that the sorts are related to each other through a type lattice. In our current implementation we do not support this type ordering.

of positive facts aligns very nicely with the notion of fitness cases in the genetic programming paradigm. An example of this technique can be seen in the results section.

The system uses the OBJ3 rewrite engine to evaluate candidate theories against the positive facts. The proof obligation arises from the theory morphism condition for the positive facts. Given a fact equation and a candidate theory, the theory morphism condition is tested by rewriting the left and right sides of the fact equation to their unique canonical forms using the equations of the candidate theory as rewrite rules. If the unique canonical forms of the left and right sides are equal then the fact equation is said to hold.

Since the equations in the candidate theories are generated at random, there is no guarantee that the theories do not contain circularities throwing the rewriting engine into an infinite rewriting loop when evaluating the facts. To guard against this situation we allow the user to set a parameter that limits the number of rewrites the engine is allowed to do per fact evaluation. This pragmatic approach proved very effective. The alternative would have been an in-depth analysis of the equations in each candidate theory adding significant overhead to the execution time of the evolutionary algorithm. In some sense this is analogous to guarding against division by zero when evaluating arithmetic expressions within the canonical genetic programming paradigm.

The fitness function used by the system to evaluate each candidate theory is

$$\text{fitness}(T) = (\text{facts}(T))^2 + \frac{1}{\text{length}(T)} ,$$

where T denotes a candidate theory, facts(T) is the number of facts or fitness cases entailed by the candidate theory, and length(T) is the number of equations in the candidate theory. The fitness function is designed to primarily exert evolutionary pressure towards finding candidate theories that match all the facts (the first term of the function). In addition, in the tradition of Occam's Razor [8] the function also exerts pressure towards finding the shortest theory that supports all the facts (second term). The system attempts to maximize this function in each generation of candidate theories.

The genetic programming engine itself is implemented as a strongly typed genetic programming system [18, 4] in the sense that it knows about the syntactic structure of theories and equations and does not have to rediscover these notions with every run. The only genetic operators we have implemented so far are fitness proportionate reproduction and a type sensitive crossover operator. We found that mutation proved too disruptive probably due to our incomplete type system implementation, as the current prototype does not properly support user declared equational logic types. This did not prevent us from performing some interesting experiments, however. We are currently working on the next generation system that supports user defined types fully.

6 EXPERIMENTS AND RESULTS

To study the system we performed three experiments with encouraging results. These experiments were inspired by case studies on the FLIP home page [2].

6.1 INFERRING STACK PROPERTIES FROM EXAMPLES

In the first example we were looking for the general concept of the stack operator *top* given a set of facts. The facts are as follows:

```
obj STACK-FACT is sort Sort .
  ops a b u v s: -> Sort .
  op top : Sort -> Sort .
  op push : Sort Sort -> Sort .
  eq top(push(v,a)) = a .
  eq top(push(push(v,a),b)) = b .
  eq top(push(push(v,b),a)) = a .
  eq top(push(push(v,u),s)) = s .
endo
```

Each ground equation in the fact theory gives a specific application instance of the operator *top*. We expect the equational inductive logic system to discover a theory that generalizes the description the operator beyond the seen instances. After 28 generations with 200 individuals the system discovered the following induced theory:

```
obj STACK is sort Sort .
  ops a b u v s: -> Sort .
  op top : Sort -> Sort .
  op push : Sort Sort -> Sort .
  vars X1 X2 X3 X4 X5 : Sort .
  eq top(push(X4,X2)) = X2 .
endo
```

This theory correctly characterizes all the ground equations in the fact theory by stating that the top of a stack is the last element pushed. The following parameters were used during this run:

Maximum number of Generations:	60
Size of Population:	200
Maximum equations for theories:	4
Maximum Rewrites:	20
Maximum depth of new individuals:	5
Maximum depth of new subtrees for mutants:	5
Maximum depth of individuals after crossover:	10
Fitness-proportionate reproduction fraction:	0.1
Crossover at any point fraction:	0.8
Crossover at function points fraction:	0.1
Number of fitness cases:	4
Selection method:	fit-prop
Generation method:	ramped
Randomizer seed:	1.0

The `Maximum Rewrites` parameter limits the number of rewrites the OBJ3 rewriting engine is allowed to perform when evaluating a fact. Readers familiar with Koza's implementation will notice that the above parameter setting does not allow for mutation.

6.2 INFERRING A RECURSIVE FUNCTION DEFINITION

In the following example we want to infer the recursive definition of the function *sum* from a set of ground equations. The fact theory is given in Peano notation where the naturals are represented as $s(0) \mapsto 1$, $s(s(0)) \mapsto 2$, *etc.* The fact theory is as follows:

```
obj SUM-FACT is sort Sort .
  op 0 : -> Sort .
  op s : Sort -> Sort .
  op sum : Sort Sort -> Sort .
  eq sum(0,0) = 0 .
  eq sum(s(0),s(0)) = s(s(0)) .
  eq sum(0,s(0)) = s(0) .
  eq sum(s(s(0)),0) = s(s(0)) .
  eq sum(s(0),0) = s(0) .
  eq sum(s(0),s(s(0)))= s(s(s(0))) .
  eq sum(s(s(0)),s(s(0)))= s(s(s(s(0)))) .
  eq sum(s(s(s(0))),s(0)) = s(s(s(s(0)))) .
  eq sum(s(s(s(0))),s(s(0))) = s(s(s(s(s(0))))) .
  eq (s(0) =/= 0) = true .
  eq (s(s(0)) =/= 0) = true .
  eq (s(s(s(0))) =/= 0) = true .
  eq (sum(s(0),0) =/= 0) = true .
  eq (sum(0,0) =/= s(0)) = true .
  eq (sum(s(0),s(0)) =/= s(0)) = true .
  eq (sum(s(0),0) =/= s(s(0))) = true .
  eq (sum(0,s(0)) =/= s(s(0))) = true .
  eq (sum(0,s(0)) =/= 0) = true
endo
```

The first half of the theory are positive facts and the second half are negative facts coded as positive facts taking advantage of OBJ3's builtin boolean operator `=/=`. As hinted at before, we take advantage of this builtin capability to express everything as positive facts rather than trying to prove that the negative facts do not hold in the induced theory. Additionally, this is more inline with the notion of fitness cases in the genetic programming engine.

After 10 generations with 200 individuals the system converged on the following theory as the induced theory:

```
obj SUM is sort Sort .
  op 0 : -> Sort .
  op s : Sort -> Sort .
  op sum : Sort Sort -> Sort .
  vars X0 X1 : Sort .
  eq sum(X1,0) = X1 .
  eq sum(X1,s(X0)) = s(sum(X1,X0)) .
endo
```

The first equation of this recursive definition of the operator *sum* states that that adding 0 to a value leaves the value unchanged. The second equation states that adding a value to the successor of another value is the same as the successor of the sum of the two values.

The parameters for the genetic programming engine in this experiment were:

```
Maximum number of Generations:                  20
Size of Population:                             200
Maximum equations for theories:                 8
Maximum Rewrites:                               25
Maximum depth of new individuals:               5
Maximum depth of new subtrees for mutants:      5
Maximum depth of individuals after crossover:   10
Fitness-proportionate reproduction fraction:    0.1
Crossover at any point fraction:                0.8
Crossover at function points fraction:          0.1
Number of fitness cases:                        18
Selection method:                               fit-prop
Generation method:                              ramped
Randomizer seed:                                1.0
```

6.3 INFERRING ANOTHER RECURSIVE FUNCTION DEFINITION

In this last example we would like to infer the concept of *even* from a set of facts. Again we use the Peano notation for naturals. The fact theory is given as follows:

```
obj EVEN-FACT is sort Sort .
  op 0 : -> Sort .
  op s : Sort -> Sort .
  op even : Sort -> Bool .
  eq even(0) = true .
  eq even(s(s(0))) = true .
  eq even(s(s(s(s(0))))) = true .
  eq (s(0) =/= 0) = true .
  eq (s(s(0)) =/= 0)= true .
  eq (s(s(s(0))) =/= 0) = true .
  eq (s(s(s(s(0)))) =/= 0) = true .
  eq (even(s(0)) =/= true) = true .
  eq (even(s(s(s(0)))) =/= true) = true .
endo
```

Please note that as in the previous example we employ the convention of coding negative examples as inequalities that must hold in the induced theory.

Unfortunately, here the system did not converge on a sensible induced theory even after as many as fifty generations with 200 individuals. We had expected something like the following:

```
obj EVEN is sort Sort .
  op 0 : -> Sort .
  op s : Sort -> Sort .
  op even : Sort -> Bool .
  var X0 : Sort .
  eq even(s(s(X0))) = even(X0) .
  eq even(0) = true .
endo
```

We suspect that the failure to converge is due to the fact that in this particular case it is paramount to distinguish between the user defined type `Sort` and the builtin type `Bool`. Due to the incomplete implementation of our type system the genetic programming engine is allowed to produce too many "junk" terms, i.e., syntactically malformed terms, which prevents the system from converging. We suspect that the system will not have any problems with this specification once we implement our type system fully.

7 CONCLUSIONS

Starting with the general notion of concept learning we developed an approach to inductive logic programming based on many-sorted equational logic with genetic programming as the underlying search paradigm. Many-sorted equational logic has a strong notion of signature and we accommodated this by developing an algebraic semantics for inductive equational logic programming using the the normal semantics for inductive logic programming as a starting point. Based on these underpinnings we implemented a prototype inductive equational logic programming system within the algebraic specification language OBJ3. Results of initial experiments looked encouraging and we expect that a more complete implementation of the type system in the prototype will remedy the current short comings.

References

[1] R. Burstall and J. Goguen. Institutions: abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, 1992.

[2] C. Ferri-Ramírez, J. Hernández-Orallo, and M.J. Ramírez-Quintana. The FLIP system homepage, 2000. http://www.dsic.upv.es/ flip/.

[3] P. A. Flach. The logic of learning: a brief introduction to inductive logic programming. In *Proceedings of the CompulogNet Area Meeting on Computational Logic and Machine Learning*, pages 1–17, 1998. http://citeseer.nj.nec.com/flach98logic.html.

[4] P. A. Flach, C. Giraud-Carrier, and J. W. Lloyd. Strongly typed inductive concept learning. In D. Page, editor, *Proceedings of the 8th International Conference on Inductive Logic Programming*, volume 1446, pages 185–194. Springer-Verlag, 1998.

[5] J. Goguen. The OBJ homepage. http://www-cse.ucsd.edu/users/goguen/sys/obj.html.

[6] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J. Jouannaud. *Software Engineering with OBJ: algebraic specification in action*, chapter Introducing OBJ. Kluwer, 2000.

[7] J. Hernández-Orallo and M. J. Ramírez-Quintana. A strong complete schema for inductive functional logic programming. In S. Džeroski and P. Flach, editors, *Proceedings of the 9th International Workshop on Inductive Logic Programming*, volume 1634, pages 116–127. Springer-Verlag, 1999.

[8] F. Heylighen. Principia cybernetica, July 1997. http://pespmc1.vub.ac.be/OCCAMRAZ.html.

[9] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.

[10] R. Ichise. Inductive logic programming and genetic programming. In H. Prade, editor, *European Conference on Artificial Intelligence*, 1998.

[11] C. J. Kennedy and C. Giraud-Carrier. An evolutionary approach to concept learning with structured data. In *Proceedings of the fourth International Conference on Artificial Neural Networks and Genetic Algorithms*, pages 1–6. Springer Verlag, 1999.

[12] J. W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 1–116. Oxford University Press, 1992.

[13] J. R. Koza. Concept formation and decision tree induction using the genetic programming paradigm. In H.-P. Schwefel and R. Männer, editors, *Parallel Problem Solving from Nature - Proceedings of 1st Workshop, PPSN 1*, volume 496, pages 124–128, Dortmund, Germany, 1-3 1991. Springer-Verlag.

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

[15] J. Meseguer and J. Goguen. Initiality, induction and computability. In M. Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–541. Cambridge, 1985.

[16] T. M. Mitchell. Generalization as search. *Artificial Intelligence*, 18(2):203–226, 1982.

[17] T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[18] D. J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[19] S. Muggleton. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995.

[20] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19/20:629–679, 1994.

[21] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[22] L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques*. The MIT Press, 1986.

[23] W. Wechler. *Universal Algebra for Computer Scientists*. Springer-Verlag, 1992. EATCS Monographs on Theoretical Computer Science, Volume 25.

[24] M. Wong and K. Leung. Genetic logic programming and applications. *IEEE Expert*, October 1995.

Machine Vision: Exploring Context with Genetic Programming

Daniel Howard and Simon C. Roberts
Software Evolution Centre
Building U50, QinetiQ
Malvern, Worcs. WR14 3PS
United Kingdom
dhoward@qinetiq.com
Phone: +44-1684-894480

Conor Ryan
University of Limerick
Limerick
Ireland

Abstract

This paper proposes an advanced method of object detection using a *data crawler*. Starting from a preliminary 'object identification', the data crawler scrutinizes the object's surroundings, and places flags where its interest has been aroused. Next, the binary map defined by these flags is analyzed statistically to quantify the context in which the object appears. The crawler's overall output indicates whether the object is a required target or a false alarm. The crawler's navigator program, its flag-placement program and its target detection program, are all controlled by a tree-based Genetic Programming (GP) method with fixed architecture Automatically Defined Functions (ADFs).

1 Introduction

The perennial problem with machine vision is to distil features which characterize an object from a huge amount of available information, i.e. the specification of an intelligent data reduction mechanism. This data reduction must be discovered because it cannot be determined *a priori*.

Moreover, if an object is viewed too closely, there is not enough contextual information upon which to make an identification. Conversely, if an object is too far away, information overload makes identification equally difficult. To deal with this surfeit of information, a way of selecting a sub-set of data, which is representative of an object, needs to be derived which allows the object to be identified.

In the context of GP, (Tackett, 1993, Poli, 1996, Daida et al., 1996, Howard and Roberts, 1999) settled for very practical schemes which manipulated statistics computed from pixel data, and (Andre, 1994, Johnson et al., 1994, Teller and Veloso, 1996, Harris and Buxton, 1996) pursued other evolvable object detection themes. (Roberts and Howard, 2000) also exploited the evolutionary paradigm to obtain the orientation of poorly defined objects such as vehicles in IR imagery. (Benson, 2000) exploited the software reuse paradigm (Koza, 1994) by embedding tree GP inside an evolvable Finite State Machine, a scheme that parallels the Automatically Defined Functions (ADFs) idea. (Roberts et al., 2001) successfully exploited problem modularities and regularities for a difficult object detection task with the subtree encapsulation idea.

This paper draws on the ADF idea to arrive at a mechanism of exploring the surrounding context and a more informed object detection (Howard et al., 2002). As post-processor to the (Roberts and Howard, 1999) scheme, it aims to intelligently lower that scheme's false alarm rate. The (Roberts and Howard, 1999) scheme processes infrared line-scan (IRLS) imagery acquired by low-flying aircraft to detect land vehicles. The scheme produces many false alarms because its aim is to detect any region which represents some part of a vehicle. Users of this scheme know that objects may be obscured by other objects accidentally or deliberately, and so a high false alarm rate is acceptable. However, the scheme cannot combine clues together to eliminate obvious misidentifications, e.g. a vehicle cannot be parked on a roof top. So the proposition is that obvious false alarms may be eliminated by exploring the context of misidentifications.

2 Overview and inspiration

It is clearly impossible to identify the vehicle object when the image is viewed from too close, for example, as shown for the vehicle detection problem in Figure 1.

Animal vision probably attempts to identify a number

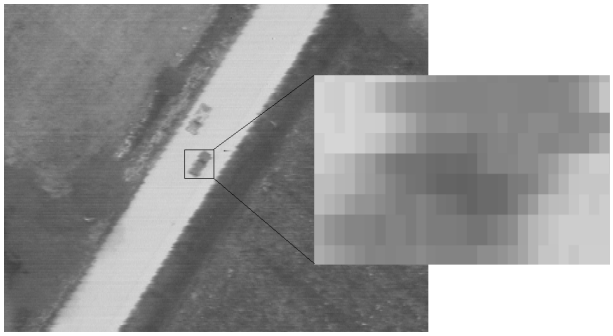


Figure 1: Two different views: close-up and from afar.

of very specific image clues that resemble some close-up detail of an object. The brain can then interpret these clues to 'imagine' the object from afar. In other words, object recognition involves constructing a mental model of an object, and superimposing this model onto the evidence perceived from the real-world. The scheme proposed in this paper is based on this idea of using both views, near and far, and of using a small number of image clues to connect these views to detect the object. The central problem then becomes how to identify the required image clues?

Evolution with GP tackles this difficult issue. Starting from the location of an object identification by the (Roberts and Howard, 1999) detector, a 'data crawler' inspects the image surroundings to discover a few useful clues. It marks a 2D binary map to flag the location of discovered 'clues', and constructs statistics based on their distribution. These statistics are also combined by GP to arrive at a decision concerning the originating pixel: whether to confirm or reject that the pixel belongs to a target object.

The evolutionary process is driven by the global objective of reducing misses and false alarms. It is implemented using GP with fixed architecture ADFs (Koza, 1994). The GP system decides the following. How should this image crawler be guided in its search for the clues? How far should it travel and turn? What should excite it sufficiently to cause it to flag the existence of an image clue?

3 Structure and representation

Each individual data crawler in the population is a GP tree structure, equipped with two result-producing branches and two ADF branches:

- First Result Branch (FRB)
- Turn Decision Branch (TDB)
- Mark Decision Branch (MDB)
- Second Result Branch (SRB)

Table 1: Function sets. GL represents glue functions.

Branch	**Functions**
FRB	$GL2$, $GL3$, MDB
TDB	+, -, *, / $(x/0 = 1)$, $\min(A, B)$, $\max(A, B)$, if $(A < B)$ then C else D
MDB	+, -, *, / $(x/0 = 1)$, $\min(A, B)$, $\max(A, B)$, if $(A < B)$ then C else D, WRITEMEM, READMEM
SRB	+, -, *, / $(x/0 = 1)$, $\min(A, B)$, $\max(A, B)$, if $(A < B)$ then C else D

FRB is allowed to call TDB and MDB many times but otherwise the branches are unrelated. SRB works on the 2D memory devised by FRB as will be explained shortly. Each branch determines a specific property of the crawler, and each has its own set of terminals and functions. Moreover, some have access to task specific working memories:

- FLAG memory
- WORKING memory
- MOVE memory

Each individual maintains these memories to save information about its past experience. Data crawler decisions, e.g. whether or not to mark the image with a flag, require a memory of past events which allows the data crawler to consult and integrate previous information prior to a decision.

FRB, working with the TDB and MDB, guides the data crawler to explore the image in the near field, and to deposit a number of 'flags' or image clues. Once this process is completed, the discriminant SRB looks from afar at this binary 2D map of 'flags' and 'no flags'. The SRB then decides whether the original starting point was correctly indicated as a true target, or whether it was a false alarm. Tables 1 and 2 give the function and terminal sets for each GP branch.

3.1 First Result Branch (FRB)

The data crawler has similarity with the Santa Fe trail ant in (Koza, 1992), e.g. the glue functions $GL2$ and

Table 2: Terminal sets (see text).

Branch	Terminals
FRB	M, TDB
TDB	μ_{11}^C, σ_{11}^C, μ_{11}^N, σ_{11}^N, μ_{11}^E, σ_{11}^E, μ_{11}^S, σ_{11}^S, μ_{11}^W, σ_{11}^W, READFLAG, READMOVE, altitude in feet (250:650)
MDB	μ_{disk}, σ_{disk}, altitude in feet (250:650), READFLAG, READMOVE
SRB	10 flag based terminals, altitude in feet (250:650), 10 textural area statistics

*GL*3 and the M or 'move terminal' to move the crawler one pixel in the direction of its travel. However, the L (left) and R (right) terminals used to turn the Santa Fe trail ant are replaced by TDB which first turns and then moves the crawler.

The FRB is executed until the data crawler has moved a predetermined number of times, i.e. $9 * w$ times, where w is the width of a vehicle. This width is automatically scaled by aircraft altitude so that w at 600ft is half w at 300ft (Roberts and Howard, 1999). FRB is iteratively evaluated until the crawler executes the predetermined number of moves. In future research the number of moves will not be predetermined but will also be allowed to evolve.

The number of image clues that could be flagged was limited to 50. When this limit was exceeded, the FRB aborted and returned the clues found so far.

3.2 Turn Decision Branch (TDB)

TDB first decides on a direction of travel and then moves the crawler in this direction. Terminals μ_{11}^C, σ_{11}^C are averages and standard deviations obtained from the pixel values in an eleven pixel square window centred on the crawler. Labels C, N, E, S, W stand for centre, north, east, south and west locations as shown in Figure 2. The window size is automatically scaled according to altitude.

TDB is always evaluated four times, once for each permutation of the order of the terminals at directions N, E, S, and W, and this produces four outputs. In the current implementation, the chosen data crawler direction is given by the permutation returning the largest output and follows certain rules.

The MOVE memory is updated after every data crawler move, i.e. when the FRB invokes M or TDB. This memory records the local turns of the crawler:

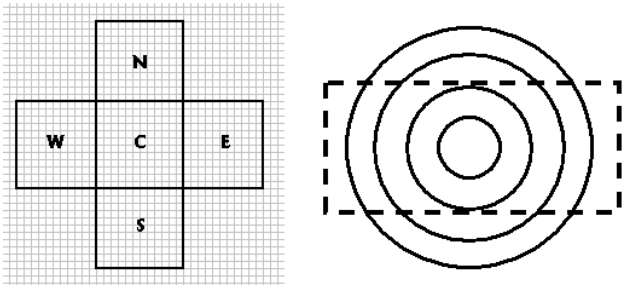


Figure 2: Left: TDB terminals: mean and standard deviation computed over square areas above, below, to the right, to the left, and centred on the pixel. These are the 'sensors' of the data crawler. Right: Four concentric pixel rings centred on a vehicle with diameters: 0.5, 1.0, 1.5 and 2.0 vehicle widths.

forward, right, back, left (F, R, B, L) rather than the absolute directions N, E, S, and W. Pixel statistics are also recorded and so the MOVE memory consists of:

1. last 10 directions, e.g. F,R,B,B,F,F,B,L,R,L is stored as [1,2,3,3,1,1,3,4,2,4];
2. last 10 μ_{11}^C;
3. last 10 σ_{11}^C;

At the start of travel, the direction memory locations are initialized to 0 and all ten values in each statistic memory location are set to the initial μ_{11}^C and σ_{11}^C.

3.3 Mark Decision Branch (MDB)

MDB determines whether a flag should be left at the current pixel position in the trail to indicate the presence of an 'image clue'. If the MDB returns a positive number it deposits a flag. This decision is based on pixel data and trail data. MDB acts as an 'IF' statement in the FRB. Placing a flag executes a THEN subtree, otherwise an ELSE subtree is executed. The executed portion is returned to FRB.

Terminals μ_{disk} and σ_{disk} in Table 2 are averages and standard deviations that are calculated over a small disk centred on the current pixel. The disk diameter is half the width of a car and corresponds to the smallest disk on the right of Figure 2. This disk is scaled by aircraft altitude, and it is distorted into an ellipse when the image is at perspective due to aircraft roll.

The FLAG memory is updated following the call to MDB. This memory consists of:

1. last 10 MDB results, e.g. [-1,-1,1,1,1,1,-1,1,-1,-1];

2. last 10 μ_{disk};
3. last 10 σ_{disk};

Here -1 stands for 'no flag', and 1 stands for 'flag set'. Initially, all values in the vector of results are set to -1 and in the other two memory vectors are set to the initial μ_{disk} and σ_{disk} respectively.

WRITEMEM and READMEM write to and read from the indexed WORKING memory, which consists of three locations or slots. All three locations are initialized to 0.0 before the crawl. Note that successive calls to MDB from FRB will not move the data crawler. The result of the repeated calls, however, can differ from one another because the memory states can change between calls. Note also that the data crawler may revisit a pixel and change its decision about flag placement.

3.4 Second Result Branch (SRB)

SRB returns a real numbered value that can be either positive (target is present) or negative (no target is present). The SRB is only executed after the FRB has completed.

The FRB uses a 2D square map to store the deposited flags. The map is initialized to 0 values and flag locations are indicated by 1 values. The suspicious pixel which is the starting point of the crawl, is at the centre of this map. If the map stores fewer than a threshold number of flags, $T_F = 4$, then the suspicious pixel is labelled 'negative' (no target present), and the SRB is not invoked. If the map stores at least T_F flags, then the SRB processes statistical measures based on the distribution of the flags.

The ten flag based terminals in Table 2 require computation of the centre of mass of the flags. They are arbitrarily based on statistical measures over the vector of distances between the centre of mass and each flag. All ten statistics are positive in value:

1. the number of flags;
2. the distance from the centre of mass to the furthest flag;
3. the longest distance between any two flags;
4. the average distance between any two flags;
5. the standard deviation in distance between any two flags;
6. the average distance between the centre of mass and each flag;
7. the standard deviation in distance between the centre of mass and each flag;
8. the degree of asymmetry of the distance distribution (skewness);
9. the relative shape of the distribution compared with a normal distribution (kurtosis);
10. the co-variance after the vector is sorted into ascending order and halved to form two sub-distributions.

Table 3: GP parameters.

Parameter	**Setting**
kill tournament	size 2 for steady-state GP
breed tournament	size 4 for steady-state GP
regeneration	90% x-over, 0% clone, 10% truncation mutation
population	500
max generations	50
max branch size	1000 nodes

Textural statistics over a wide area are input to SRB so that the branch can form its own image segmentation. In this way, the SRB can give different detections for the same flag distribution appearing in different textural contexts. For example, a horse in a rural area may receive the same flag distribution as a vehicle in an urban area, but the SRB could discriminate the horse as a false alarm, whilst detecting the vehicle, due to the differences in rural and urban textures (Roberts and Howard, 1999). These textural statistics are based on a co-occurrence matrix and are taken over an area of 5 car-lengths square.

4 GP implementation

The GP run parameters are given in Table 3. Crossover was branch typed, meaning that it could only exchange genetic material between like branches, e.g. an FRB only with another FRB. Each branch was assigned a probability to participate in crossover. The FRB had a probability of 40% and the other three branches had a probability of 20%. Truncation mutation selects any node (and the associated subtree) and replaces it with a terminal from the relevant terminal set.

The suspicious points detected by the (Roberts and Howard, 1999) scheme constituted the fitness cases. These points were known to be 'true positives' (TP) or 'false positives' (FP), i.e. false alarms. The fitness

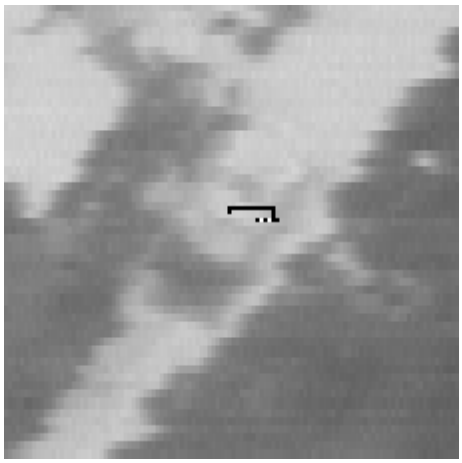


Figure 3: Trail terminating at the right of a vehicle.

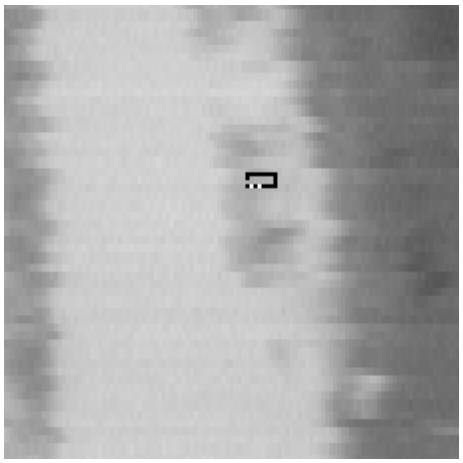


Figure 4: Trail terminating at the left of a vehicle.

measure was computed once all fitness cases had been processed:

$$\text{fitness} = \frac{\alpha TP}{\beta FP + \text{nVehicles}}$$

The parameters α and β balance the importance of TP relative to FP. These parameters were optimized to minimize FP whilst retaining near-maximal TP.

5 Experimental Results

The scheme was implemented to process previously detected vehicles and false alarms in airborne reconnaissance IRLS imagery (Roberts and Howard, 1999). The vehicles appear in many sizes and orientations, in many perspectives and thermal states, next to various objects (such as buildings) which cast thermal shadows onto the vehicles, and in many environments and weather conditions. Hence, the vehicle detection task in these operational images is extremely challenging.

This section presents examples of the trails produced by a data crawler evolved with $\alpha = 2.6$ and $\beta = 1.0$. The trails are drawn on sub-images cropped from IRLS imagery with approximate dimensions of $3000 \times 10,000$ pixels. The end of each trail is indicated with two dots, except when the trail is iterative.

Figures 3 and 4 show that the crawler can make a similar trail on different vehicles. Each trail starts on the vehicle's roof because this was the starting point as detected by the (Roberts and Howard, 1999) scheme. Interestingly however, each trail terminates at the vehicle's side, even though the vehicles are oriented differently.

Generally, when the crawler is applied to the *same* vehicle, but with different starting points, it can take different routes but still detect the vehicle. Figure 5 displays a case where the crawler homed in on a vehicle's wind-screen.

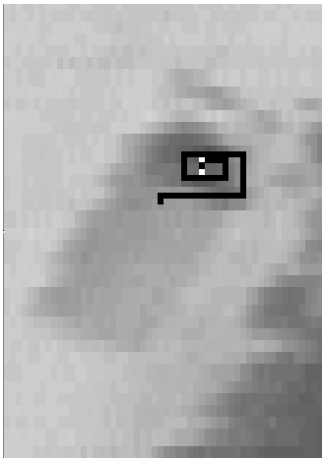


Figure 5: Spiral trail on a vehicle.

The power of the data crawler is illustrated in Figures 6 and 7, where false alarms are rejected due to contextual information. In other words, the 'false positives' as detected by the (Roberts and Howard, 1999) scheme are correctly converted to 'true negatives'. In Figure 6, the starting point could resemble some feature of a vehicle, but crawling onto an open roof dismissed this hypothesis. Similarly, a more complicated trail in Figure 7 rejects a grass verge by crawling onto an open carriageway. The grass verge was probably initially detected as a false alarm because it has the width of a car. This sub-image clearly shows the jitter in the IRLS imagery which hinders the detection task. Experiments reduced the false alarm rate typi-

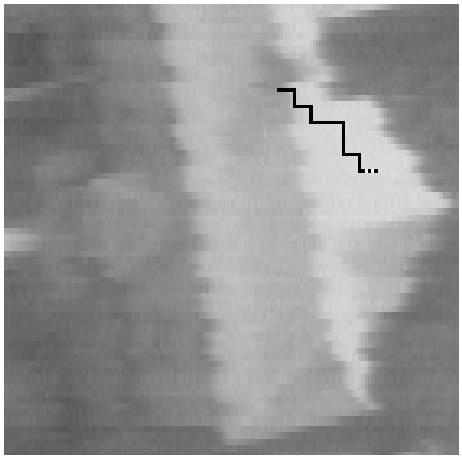


Figure 6: A potential vehicle feature on a roof is correctly rejected.

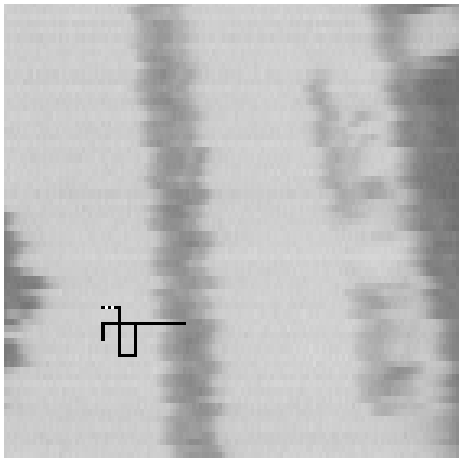


Figure 7: Trail commencing on a grass verge between carriageways. Note the two cars on the right.

cally by 30%.

Other false alarms are rejected not by wandering trails, but instead by tight trails which stay close to the starting point, as shown in Figures 8 to 10. In these cases, the starting point is within the confines of an object which is too small to be a target, and thus there is no need to explore the object's surroundings. Note that the ends of these trails are not shown because the trails are iterative, with the crawler retracing its steps or performing a cyclic motion.

Systematic flag formations to characterize object detections are not visibly evident in the crawler's trails. For example, the false alarms are not characterized by a typical flag distribution pattern. This is probably because the flag placement, and indeed the trail itself,

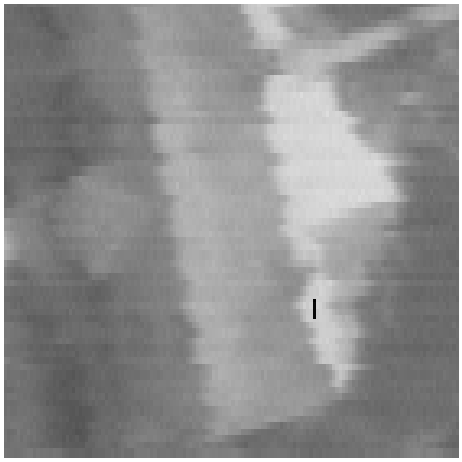


Figure 8: Iterative trail on an obstructed section of roof.

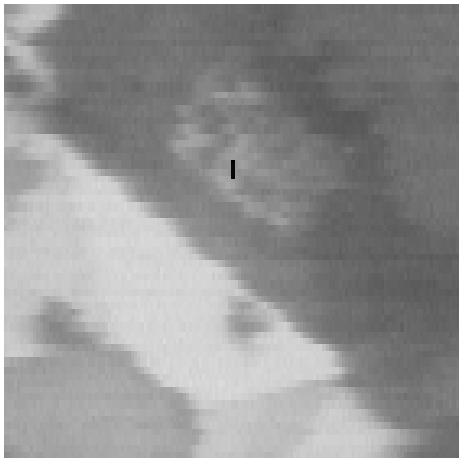


Figure 9: Iterative trail on a garden plot.

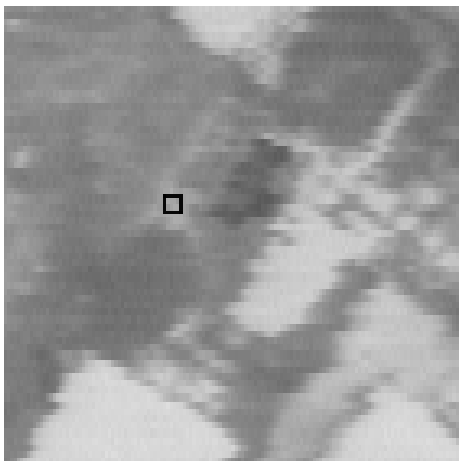


Figure 10: Cyclic trail in a rural driveway near a tree and out-buildings.

depends on the particular 'image clues' in an object's vicinity.

However, some typical flag distribution patterns are evident. There is a tendency for the crawler to place flags with a near-equal distribution along the trail, and to prefer placements on turning points. More strikingly, the last few steps in each trail are often flagged. The crawler governs the length of its trail by iteratively retracing its final steps to effectively time-out the FRB. Hence, these final steps are often *repeatedly* flagged.

Recall that at least $T_F = 4$ flags had to be set for the SRB to be executed, and that fewer flags gave a default negative detection (Section 3.4). Experiments found that the number of individuals that place at least T_F flags increased during each evolution run. This suggests that individuals could not use the number of flags alone to detect the vehicles, by simply invoking SRB to output a positive value. This also suggests that SRB was employed to manipulate the flag distribution statistics in a more sophisticated way.

For example, each step in the iterative trails in Figures 8 and 9 are flagged, thus yielding short distances for the flag distribution statistics. The SRB could then interpret these statistics as an indicator to reject these false alarms.

6 Conclusions

This paper describes a data crawler to improve target detection by exploring the context in which candidate targets appear. The crawler initially processes *close-up* views to "sense" an object's surroundings and to flag where it becomes aroused. The distribution of these flags then represents a *distant* view of the object, where contextual information has necessarily been greatly reduced. The crawler then indicates whether the object is a required target or a false alarm. All aspects of the crawler's design are evolved using GP with fixed architecture ADFs.

The data crawler improved the performance of a preliminary target detection scheme, by typically reducing the false alarm rate by 30% whilst retaining most of the actual targets.

The present scheme is computationally demanding, due to the calculation of pixel statistics every time the crawler moves. Computation speed may be improved by allowing the crawler to jump, and indeed this may be more in-keeping with the way that a scene is scanned in animal vision. Other advancements could explore iterations between the SRB and FRB to reinforce detection decisions, and "colonies" of crawlers could participate via pheromone trails.

References

[Andre, 1994] David Andre (1994). Automatically Defined Features: The Simultaneous Evolution of 2-Dimensional Feature Detectors and an Algorithm for Using Them. In Kenneth E. Kinnear, Jr. (ed), *Advances in Genetic Programming, 477–494*, MIT Press.

[Benson, 2000] Karl A Benson (2000). Evolving Finite State Machines with Embedded Genetic Programming for Automatic Target Detection within SAR Imagery. In *Proceedings of the 2000 Congress on Evolutionary Computation CEC00, 1543–1549*, IEEE Press.

[Daida et al., 1996] Jason M. Daida, R. G. Onstott, T. F. Bersano-Begey, S. J. Ross and J. F. Vesecky (1996). Ice Roughness Classification and ERS SAR Imagery of Arctic Sea Ice: Evaluation of Feature-Extraction Algorithms by Genetic Programming. In *Proceedings of the 1996 International Geoscience and Remote Sensing Symposium, 1520–1522*, IEEE Press, Washington.

[Harris and Buxton, 1996] Christopher Harris and Bernard Buxton (1996). Evolving Edge Detectors with Genetic Programming. In Koza, Goldberg, Fogel and Riolo, *Genetic Programming 1996: Proceedings of the First Annual Conference, 309–315*, MIT Press.

[Howard et al., 1999] Daniel Howard, Simon C. Roberts and Richard Brankin (1999). Evolution of Ship Detectors for Satellite SAR Imagery. In Poli, Nordin, Langdon and Fogarty (eds), *Genetic Programming, Proceedings of EuroGP 1999, 135–148*, Lecture Notes in Computer Science 1598, Springer-Verlag.

[Howard and Roberts, 1999] Daniel Howard and Simon C. Roberts (1999). A Staged Genetic Programming Strategy for Image Analysis. In Banzhaf, Daida, Eiben, Garzon, Honavar, Jakiela and Smith (eds), *Proceedings of the Genetic and Evolutionary Computation Conference, 1047–1052*, Morgan Kaufmann.

[Howard and Roberts, 1999b] Daniel Howard and Simon C. Roberts (1999). Evolving Object Detectors for Infrared Imagery: a Comparison of Textural Analysis Against Simple Statistics. In Miettinen, Mäkelä and Toivanen (eds), *Proceedings of EUROGEN99 - Short Course on Evolutionary Algorithms in Engineering and Computer Science, 79–86*, Dept. of Mathematical I. T. Collections No. A 2/1999, University of Jyväskylä, Finland, ISBN 951-39-0473-3.

[Howard et al., 2002] Daniel Howard, Simon C. Roberts and Conor Ryan (2002). The BORU Data Crawler for Object Detection Tasks in Machine Vi-

sion. *Applications of Evolutionary Computing: European EC Workshop 2002, Kinsale, Ireland, 222–232*, Lecture Notes in Computer Science 2279, Springer-Verlag.

[Johnson et al., 1994] Michael Patrick Johnson, Pattie Maes and Trevor Darrell (1994). Evolving Visual Routines. In Rodney A. Brooks and Pattie Maes (eds), *ARTIFICIAL LIFE IV, Proceedings of the fourth International Workshop on the Synthesis and Simulation of Living Systems, 198–209*, MIT Press.

[Koza, 1992] John R. Koza (1992) Genetic Programming, MIT Press.

[Koza, 1994] John R. Koza (1994) Genetic Programming II: Automatic Discovery of Reusable Programs, MIT Press.

[Poli, 1996] Riccardo Poli (1996). Genetic Programming for Image Analysis. In Koza, Goldberg, Fogel and Riolo (eds), *Genetic Programming 1996: Proceedings of the First Annual Conference, 363–368*, MIT Press.

[Roberts and Howard, 1999] Simon C. Roberts and Daniel Howard (1999). Evolution of Vehicle Detectors for Infrared Linescan Imagery. In Poli, Voigt, Cagnoni, Corne, Smith and Fogarty, *Evolutionary Image Analysis, Signal Processing and Telecommunications: First European Workshop, EvoIASP 1999 and EuroEcTel 1999, 110–125*, Lecture Notes in Computer Science 1596, Springer-Verlag.

[Roberts and Howard, 2000] Simon C. Roberts and Daniel Howard (2000). Genetic Programming for Image Analysis: Orientation Detection. In Whitley, Goldberg, Cantu-Paz, Spector, Parmee and Beyer, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000), 651–657*, Morgan Kaufmann.

[Roberts et al., 2001] Simon C. Roberts, Daniel Howard and John R. Koza (2001). Evolving Modules in Genetic Programming by Subtree Encapsulation. In Miller, Tomassini, Lanzi, Ryan, Tettamanzi and Langdon (eds), *Genetic Programming, Proceedings of EuroGP 2001, 160–175*, Lecture Notes in Computer Science 2038, Springer-Verlag.

[Tackett, 1993] Walter Alden Tackett (1993). Genetic Programming for Feature Discovery and Image Discrimination. In Stephanie Forrest (ed), *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93, 303–309*, Morgan Kaufmann.

[Teller and Veloso, 1996] Astro Teller and Manuela Veloso (1996). PADO: A New Learning Architecture for Object Recognition. In Katsushi Ikeuchi and Manuela Veloso, *Symbolic Visual Learning*, Oxford University Press.

Genetic Programming and Multi-Agent Layered Learning by Reinforcements

William H. Hsu
bhsu@cis.ksu.edu
Department of Computing and Information Sciences
Kansas State University
Manhattan, KS USA 66506-2302

Steven M. Gustafson
smg@cs.nott.ac.uk
School of Computer Science and Information Technology
University of Nottingham
Jubilee Campus, Nottingham UK NG8 1BB

Abstract

We present an adaptation of the standard genetic program (GP) to hierarchically decomposable, multi-agent learning problems. To break down a problem that requires cooperation of multiple agents, we use the *team objective function* to derive a simpler, intermediate objective function for pairs of cooperating agents. We apply GP to optimize first for the intermediate, then for the team objective function, using the final population from the earlier GP as the initial seed population for the next. This *layered learning* approach facilitates the discovery of primitive behaviors that can be reused and adapted towards complex objectives based on a shared team goal. We use this method to evolve agents to play a subproblem of robotic soccer (keep-away soccer). Finally, we show how layered learning GP evolves better agents than standard GP, including GP with automatically defined functions, and how the problem decomposition results in a significant learning-speed increase.

1 INTRODUCTION

For complex problems with low-level primitive operations, such as robotic soccer [Ki97, MNH97], it is intractable to search for a direct solution using genetic programming (GP). This is due in part to the combinatorial explosion of the GP search space as a function of the problem state space – e.g., the size of the playing field. [SVR99] Other factors, such as operator granularity, also contribute to this growth. Many of GP researchers who have worked on robotic soccer have simplified the GP search space through problem redefinition: raising the level of terminals in order to evolve higher-level behaviors [Lu98] or using a more sophisticated fitness function [AT99]. Because robotic soccer is a multi-agent system (**MAS**) problem that is based upon a real game played by humans, it is helpful to compare learning strategies with those of human teams, even if we use a different approach to automatically develop a solution. One important observation is that the structure of team training in real soccer involves individual, pair, and small group drills, resulting in a well-defined hierarchy of behaviors. Traditional GP produces hierarchical programs by evolving and reusing automatically defined functions (ADFs). [Ko94, RB94]

In this paper, we show how *layered learning* can also achieve reuse – faster and more reliably than GP with ADFs – in developing a solution to an MAS subproblem of robotic soccer. Just as ADFs provide reusable code and subroutine structure [Ko94], layered learning provides a way to build solutions using a divide-and-conquer approach [St00, SV00a]. The difference between ADF learning and layered learning, using GPs or other methods, is that layered learning describes a way to *train* a learning intelligent agent, while ADFs describe a way to *implement structure* in the agent representation – i.e., code.

Layered learning GP (**LLGP**) [GH01] can be used to break down MAS learning tasks by first evolving solutions for smaller fitness cases or for smaller groups of agents with a more primitive fitness criterion. While our adaptation of layered learning to GP is based in part upon Stone and Veloso's work in reinforcement learning [SV00a], similar approaches have been developed that perform sequential evolution of populations using different fitness functions [De90, HHC94].

This paper extends our previous study of LLGP for an MAS task in the robotic soccer domain [GH01] with further experiments and analysis of LL behavior. We focus on automatic tuning and validation of *intermediate representations* in incremental LL. The purpose of our test bed is to facilitate development of fitness criteria for "coaching" or training agents based upon their strictly cooperative performance in a two-agent task. We then use the evolved individuals to seed a population of agents to be further improved in three-way competitive interaction against a fourth agent, the opponent. This new population and the associated GP form the second layer of the LLGP system. The product of LLGP is an agent that is *evolved* using highly fit primitive agents, but does not necessarily contain exact copies of these primitive agents as subroutines.

Another advantage of layered learning is that it provides a logical methodology for implementing a hierarchical approach to teamwork. In order to evolve more complex teamwork, we may be able to take advantage of the dependency of behaviors involving three or more teammates upon primitive behaviors involving just two. For example, a low-level primitive in soccer is passing the ball, a two-agent activity that is incorporated into several multi-agent activities: guarding the ball; moving the ball downfield; setting up for a goal attempt; etc. In the rest of this paper, we shall explore LLGP for MAS problems using *keep-away soccer*, a subproblem of robotic soccer [SV00a, GH01] that shows how complex teamwork can be hierarchical in nature and therefore can be learned efficiently in a hierarchical fashion.

2 THE KEEP-AWAY SOCCER DOMAIN

2.1 DEFINITION AND JUSTIFICATION

We call *keep-away soccer* the task of keeping the ball away from a defensive player who is attempting to capture it from multiple offensive opponents. We chose keep-away soccer as a learning test bed for MAS because it:

1. captures a compositional element of teamwork, composing and refining passing behaviors to achieve full keep-away soccer behavior, that occurs in real and robotic soccer
2. elides some objectives of soccer (such as moving the ball downfield and attempting to score) that, while crucial, would overcomplicate our study of basic low-level MAS
3. allows us to easily adjust opponent difficulty

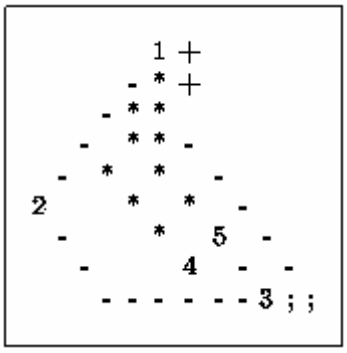


Figure 1. Screen capture of simulator. 1, 2, and 3 are offensive agents, 4 is the defender, and 5 is the ball, which moves in trajectory 3-1-2-3-1.

Although there is a strong compositional element, learning to pass the ball effectively is only part of the keep-away soccer learning task. In real soccer, human players learn to minimize the number of turnovers to the offensive opponent by passing accurately, move to receive a pass, and make themselves open to receive a pass, and control the ball effectively. For 3 or more agents to coordinate effectively, each must be able, when in possession of the ball, to: select a teammate to pass to, time the pass appropriately, and maintain open at least one passing lane.

Figure 1 shows a text-mode screen capture from the simple program that we used to visualize and animate games of keep-away soccer. The figure depicts three offensive agents passing the ball in a counterclockwise motion (agent 3 passes the ball twice) about a defender. The trail of the ball is denoted by '-'. The symbols '+', ';', and '*' show the paths of agents 1, 3, and 4, respectively. The simulation and visualization were run for about 30 time steps to collect the screen capture.

Several MAS variants of robotic soccer exist; keep-away soccer belongs to the category of multi-agent learning with *homogeneous, noncommunicating* agents [SV00b] – those that share identical code but have no direct channels of communication other than by observing the behavior of teammates. This type of problem requires more robust, autonomous solutions and is therefore an interesting framework for teamwork learning.

Soccer, whether analyzed as a human or robotic game, can be broken down into skill-optimization subproblems such as ball control, passing, and moving. Keep-away soccer can be decomposed in the same manner. A natural way to reduce complex, MAS problems, such as that investigated in keep-away soccer, could generalize to other cooperative MAS problems [Ta97].

2.2 PROBLEM SPECIFICATION

Test beds for robotic soccer-playing agents have been framed through the *RoboCup* competition [KAK+95, As99]. These have been found to be rich experimental environments for many MAS research areas, including flexible teamwork learning [TAA+99], and methodologies, including hierarchical sensing and reinforcement learning by *Q*-learning, temporal differences, team-partitioned algorithms [SV98], artificial neural networks, and genetic programming [As99]. At present, however, hand-coded and hybrid learning techniques that employ a large amount of hand-coded domain-specific knowledge still outperform strategies that are learned automatically.

In keep-away soccer, three offensive agents are located on a rectangular field with a ball and a defensive agent. The defensive agent moves twice as quickly as the offensive agents, and the ball, when passed, moves twice as quickly as the defensive agent. This is similar to the predator-prey problem in [LS96], where more than one agent is required to solve the problem. The objective in keep-away soccer is to minimize the number of times the ball is turned over to the defender. A turnover occurs at every discrete time step in which the defender is within one grid unit of the ball. Thus, subsidiary objectives for offensive agents are to continuously move and pass the ball to one another in order to minimize turnovers.

We think of keep-away soccer as consisting of two layers of behavior: passing accurately with no defensive agent present, and moving and passing with a defender to minimize the number of turnovers that occur during a game. The two layers of behaviors come from a human-like view of soccer, but are not heavily dependent upon domain knowledge. Both types of behavior are important to playing good keep-away soccer, but the operational definition does not necessarily give us a way to measure the effectiveness of a team of agents who have just played keep-away soccer, which would be useful for finding a fitness function.

Next, we present the application of layered learning to GP and explain further how the keep-away soccer is a good, illustrative test bed for LLGP.

3 LAYERED LEARNING

Layered learning is a term used in the machine learning and intelligent agents literature [St00, SV00a] to describe a task-driven and often incremental approach to acquiring hierarchies of behaviors by reinforcement learning.

de Garis [De90] introduced a very similar concept that he called *behavioral memory* for a genetic algorithm that encoded neural networks. Weights and signs of the networks evolved for one behavior were used to construct a new population, evolved for a second behavior. Some of the initial network persisted in the solutions for the new behavior. Schoenauer and Xanthakis [SX93] then later applied this concept for constrained genetic algorithm optimization.

Harvey *et al* [HHC94] used a layered, incremental learning approach to robot control in a vision-based navigation system. The authors achieved this by sequentially evolving a population using a range of targets from simple to complex. Winkeler and Manjunath [WM98] and Eriksson [Er00] later analyzed this approach toward incremental learning.

Dorigo *et al* [DC97] developed another hierarchical learning system that is somewhat different from layered learning as we have adapted it. In this method, inputs and processing elements are organized into a hierarchy (from simple to complex), each of whose layers is incrementally trained and frozen. This is similar to previous work applied in domains such as robot soccer, but is not identical to layered learning or behavioral memory as these methods do not arrest learning in a particular portion of the hierarchical model.

Applying the layered learning paradigm to a problem consists of breaking that problem up into a hierarchy of subproblems. The original problem is then solved sequentially, by using the learning results from all the member problems of each layer in the next layer. This is conceptually similar to many other divide-and-conquer learning paradigms, but a key difference is that the structure of the *solution* does not necessarily reflect this procedural hierarchy of *training*. For example, programs evolved for a subtask in LLGP are used to seed an initial population for the next layer, but they may not be incorporated verbatim in the overall solution as ADFs are. This type of hierarchical solution is different from the type that ADF-based GP learning proposes to find, which focuses on code reuse and structure rather than on how the subtasks are learned.

Problems that attempt to achieve human-competitive behaviors [Ko98], such as robotic soccer and keep-away soccer, lend themselves well to bottom-up decomposition. This is because human task learning, especially of cooperative multi-agent behavior, often occurs in a bottom-up fashion where individuals or small groups first learn smaller tasks, then how to compose and coordinate them to solve larger tasks. When the problem is of this type and we are already using a biologically motivated method such as GP, it seems very natural to use a bottom-up decomposition of the problem that simulates this aspect of human learning and allows GP to learn each of the smaller problems.

Table 1 is a variant of the table found in [SV00a], which we have adapted to correlate each prerequisite of layered learning with a property of genetic programming for keep-away soccer.

Table 1: Requirements for using layered learning and GP keep-away soccer justifications.

Layered Learning	**Genetic Programming**
1. Learning from raw input is not tractable	Complex MAS problems for GP need to be defined at multiple levels ✓
2. A bottom-up decomposition is given	MAS learning task is compositional ✓
3. Learning occurs independently at each level	GP can be applied to each layer independently ✓
4. The output of one layer feeds the next layer's input	The population in the last generation of one layer is used as the next layer's initial population ✓

When we modify standard GP for layered learning, we need to develop a learning objective for each layer, i.e., the fitness at each layer that selects ideal individuals *for the subtask*. As seen in [Lu98], using a single-objective fitness value often leads to the best performance, and is much easier than trying to define multi-objective fitness functions. While multi-objective fitness functions should allow GP to evolve more complex behaviors, it becomes more difficult to decide what the components of fitness should be and how important each one is to the solution. In preliminary experiments, we found that it was infeasible to develop either a set of Pareto optimization criteria or a weighted function over multiple objectives for keep-away soccer. Instead, we chose to focus on

automatically discovering how to compose *passing agents* into *keep-away soccer agents*.

Another issue we addressed for layered learning in GP is the transfer of the population from the last generation of previous layer to the initial population of the next. The ideal team will consist of individuals with high fitness on the coordinated MAS task. Meanwhile, in every population, there are certain individuals that have a better fitness than others. We might therefore consider copying that best individual only and seeding the entire initial population of the subsequent layer with it. However, this duplication removes the diversity that was evolved in the previous layer, which may be detrimental because the best individual on the subtask may be a suboptimal problem solver for the overall coordinated team activity. Thus, we designed two experiments using LLGP: one that duplicates the best individual and one that simply copies the entire population.

The final issue we address for LLGP is learning-speed improvement: to what degree can layered learning simplify the learning problem, allowing the target fitness to be reached faster than with standard GP? This increase in the slope of the learning-speed curve [Ka95] is to be distinguished from *speed-up learning*, wherein the efficiency of the learned problem solver is improved. We show how layered intermediate and team fitness objectives achieve greater learning-speed than a monolithic fitness objective in the keep-away soccer test bed. We also demonstrate a technique for empirically choosing a point at which to stop learning primitive MAS behaviors and switch to the high-level MAS behavior.

4 GP AND EXPERIMENT DESIGN

We designed four initial GP experiments to investigate and benchmark the performance of LLGP: standard GP (**SGP**), GP with ADFs (**ADFGP**), LLGP with the best individual duplicated to fill initial populations (**LLGP-Best**), and LLGP with the entire final population of the first layer used to seed the next (**LLGP-All**). SGP and ADFGP use the single *monolithic* (i.e., non-layered) fitness function of minimizing the number of turnovers that occur in a simulation. ADFGP allows each tree for kicking and moving to have two additional trees that represent ADFs, where the first ADF can call the second, and both have access to the full function set available for SGP. LLGP-Best and LLGP-All both have two layers; the fitness objective for the first layer is to maximize the number of accurate passes (a two-agent task evaluated over teams of three copies of the same individual, on the same size field as the keep-away soccer task), while fitness objective for the second layer is to minimize the number of turnovers.

We developed two variations on each experiment, with maximum generation values of 51 and 101. The stopping criterion for both variations is achieved when an ideal fitness measure of 0 (where fewer turnover turns are better) is found, or the maximum generation is reached. Our preliminary experiments indicated that a population size of 2000 yielded good results for the keep-away soccer domain using both SGP and ADFGP. We also found that the 101-generation SGP achieved better convergence in fitness and individual size and the 51-generation SGP, with negligible fitness improvement after 101 generations.

The genetic crossover operator generates 90 percent of the next generation; tournament selection generates the other 10 percent. [Ko92] The tournament size is 7, with maximum depth 17. Table 2 summarizes the terminal set used, consisting of vectors that are egocentric, or relative to the agent whose tree is being evaluated. Table 3 summarizes the function set used, where all functions operate on and return vectors. Both sets are similar to those used in [Lu98] and [AT99].

Table 2: Keep-away soccer terminals (egocentric vectors)

Terminal	**Description**
Defender	Vector to opponent
Mate1	Vector to first teammate
Mate2	Vector to second teammate
Ball	Vector to ball

Table 3: Keep-away soccer function set

Function (arguments)	**Description**
Rotate90(1)	Rotate current vector 90 degrees counter-clockwise
Random(1)	New random vector with magnitude between 0 and current value
Negate(1)	Reverse vector direction
Div2(1)	Divide vector magnitude by 2
Mult2(2)	Multiply vector magnitude by 2
VAdd(2)	Add two vectors
VSub(2)	Subtract two vectors
IFLTE(4)	if $\|\mathbf{v_1}\| < \|\mathbf{v_2}\|$ then $\mathbf{v_3}$ else $\mathbf{v_4}$

The GP system we use was developed by Luke and is called Evolutionary Computation in Java (ECJ) [Lu00]. The simulator we developed for keep-away soccer abstracts some of the low-level details of agents playing soccer from the *TeamBots* [Ba01] environment, which in turn abstracts low-level details from the *SoccerServer* [An98] environment. Abstractions of this type allow the keep-away soccer simulator to be incorporated later to learn strategies for the *TeamBots* environment and *SoccerServer*.

In *SoccerServer* and *TeamBots*, players push the ball to maintain possession. To kick the ball, the player needs to be within a certain distance. For keep-away soccer, we eliminate the need for low-level ball possession skills and allow offensive agents to have possession of the ball. Once an agent has possession, it can only lose possession by kicking the ball, i.e., by evaluating its kick tree. Because we use vectors that have direction and magnitude, this implementation would allow for dribbling actions to be learned, where the agent simply passes the ball a few units away. This abstraction greatly simplifies the problem and still allows for a wide range of behaviors to be learned.

At each simulation step that allows agents to act, if the agent has possession of the ball – i.e., the agent and ball occupy the same grid position – the agent's kick tree is evaluated. The kick tree evaluates to a vector that gives the direction and distance to kick the ball. Otherwise, the agent's move tree is evaluated. Both trees are composed of terminals listed in Table 2 and functions listed in Table 3.

For layered learning experiments, the first 5-50 percent of the maximum number of generations are spent in Layer 1 learning accurate passing without a defender present. To evaluate accurate passes, we count the number of passes that are made to a location within 3 grid units of another agent. The fitness function for this *intermediate objective* is then (200 – *passes*), where there are 200 time steps per simulation; a fitness of 0 is best and one of 200 is worst. The remaining 50-95 percent of the generations are spent in Layer 2 with a fitness value that is inversely proportional to the number of turnovers that occur with a defender present. This is the *team objective*. The defender uses a hand-coded strategy, based upon one of the standard *TeamBots* [Ba01] defensive agents, that always moves towards the ball to cause a turnover.

Each evaluation of an individual in the simulator takes 200 time steps, where the ball can move on each step, the defender moves on every other time step, and all offensive agents move together on every fourth time step. The initial configuration of the simulation places the defensive agent in the center of a 20-by-20 unit grid. The field is then partitioned into three sections: the top half and the bottom left and right quadrants. One offensive agent is placed randomly in each section, and the ball is placed a few units from one of the offensive agents, chosen at random.

Early runs of the system resulted in local optima being achieved; the most common of these was a control policy in which all offensive agents crowded the ball to prevent a defender from stealing it, causing turnover. To eliminate this "loophole", the defender, if blocked from the ball, can move *through* an offensive agent without the ball by simply trading places with the opponent if the two are adjacent on the grid.

5 RESULTS

Each experiment was run 10 times, and averages were taken across the runs. For all experiments, we achieved the best convergence behavior with 100 generations, so this was used as the baseline for SGP, ADFGP, and all LLGP variants.

Table 4 shows our initial experimental results. For ADFGP, Good-Average represents the average of the 10 best runs selected from among 20. ADFGP experiments converged to two clusters of fitnesses – one better than SGP, the other much worse. When we considered the individual size of the good cluster, we found that the poor cluster contains individuals with about half the number of nodes as individuals in the good cluster. Prefiltering ADFGP runs based upon individual size may be an appropriate remedy, but this is beyond the scope of this paper, as we are focusing on LLGP. We report both overall and good averages here, however, to show that LLGP can achieve performance as high as the good cluster's.

As shown in Table 4, our first LLGP experiment divided 101 generations into 40 for Layer 1 (successful pass criterion) and 61 for Layer 2 (minimum turnover criterion). Copy-Best represents the LLGP-Best seeding method for Layer 2; Copy-All, the LLGP-All method. These initial results did not indicate any notable advantage or disadvantage of LLGP, indicating only that we can obtain comparable solutions using LLGP-All, SGP, and ADFGP.

Table 4: Results for experiments with population size = 4000, max generations = 101, averaged over 10 runs. *Lower f (anti-fitness) values are better.*

	SGP	**ADFGP**		**LLGP, 40-61**	
		Avg.	**Good-Avg.**	**Copy-Best**	**Copy-All**
Best *f* gen. 101	11.25	19.67	8.75	23.71	12.67
Mean *f* gen. 101	66.89	60.21	64.27	82.03	64.64
Avg. ind. sz. gen. 101	228.74	113.25	123.07	161.71	171.40
First gen. *f* ≤ 20	33	62	22	101	55
Best *f* of run	9.0 ± 4.98	16.56 ± 17.45	6.83	19.29	9.0 ± 2.73

Table 5: Results for different Layer 1 durations (population size = 2000), averaged over 10 runs. *Lower f (anti-fitness) values are better.*

Layer 2 Start Generation	First Gen. $f \leq 20$	First Gen. $f \leq 15$	Best f Gen. 101	Best f of Run
5	62	79	12	11.75
10	18	30	11.4	9.7
15	24	43	9.63	9.38
20	38	47	9.6	9.6
25	47	80	11.88	11.25
30	51	58	14.1	12.7
35	46	55	7.75	7.25
40	57	82	13.6	13.2
45	67	93	14.11	13.11
50	62	82	11.5	11.1

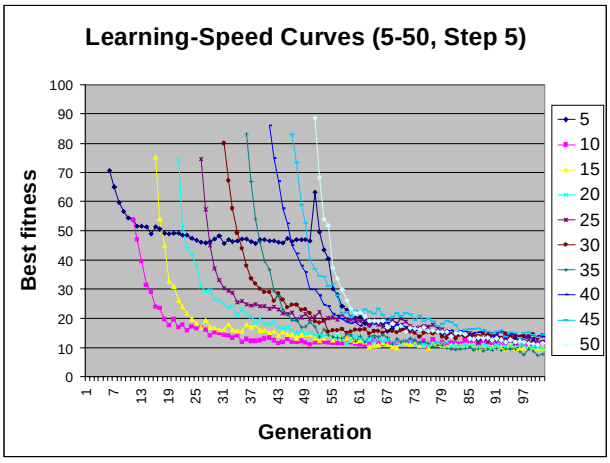


Figure 2. Layer 2 learning-speed curves (each starting at the end of Layer 1) for different Layer 1 durations. Lower is better.

We hypothesized that we were not yet realizing the full improvement in learning-speed that could be achieved using LLGP. To test this hypothesis, we plotted the Layer 2 learning-speed curves [Ka95] shown in Table 5 and Figure 1 for the following LLGP-All configurations: 5 generations in Layer 1 and 96 in Layer 2, 10 and 91, up to 50 and 51. The 10-91, 15-86, and 20-81 versions of LLGP-All achieve better convergence than those that start Layer 2 later, except for 35-66. Even accounting for the "early start", we can see that the convergence rate is faster and the final fitness is better for LLGP when Layer 1 lasts between 10 and 20 generations. We ran a second series of Layer 2 learning-speed curves (6 through 15, step 1) that indicated that the learning rates for 10 through 15 were not significantly different. We have not yet evaluated the inherent benefit to generalization quality – i.e., overfitting control and reusability – of stopping Layer 1 earlier, though this may be a good question for future experimentation.

A population size of 2000 is used for the fitness curves, as the performance for 2000 is similar to that for 4000, as reported in [Gu00]. Note that if the learning-speed curve for Layer 1 duration of 0 were plotted in Figure 2 above, it would be equivalent to that of the SGP, because the SGP runs for 101 generations with only the team objective function (Layer 2 fitness).

Table 6: Results for experiments with population size = 4000, max generations = 101, averaged over 10 runs. *Lower f (anti-fitness) values are better.*

	SGP	Good-ADFGP	LLGP-All, 10-91
Best f gen. 101	11.25	8.75	9.43
Mean f gen. 101	66.89	64.27	70.39
Avg. ind. sz. gen. 101	228.74	123.07	249.21
1st gen. $f \leq 20$	33	22	26
Best f of run	9.0 ± 4.98	6.83	*5.78 ± 2.28*

Having found that the 10-91 LLGP exhibited a better learning speed curve, we repeated the LLGP-All experiment with population size 4000 and found that it was able to match the Good-ADFGP performance, converged at least as quickly as any other GP, and resulted in the lowest best-of-run fitness values we found (fewer than 6 turnovers per simulation). This result is shown in Table 6, with the SGP and Good-ADFGP results repeated for comparison. We note that the Layer 2 individuals are much larger for LLGP-All-10-91 than for LLGP-All-40-61. That is, while stopping Layer 1 early yields a slight improvement in overall fitness and a significant improvement in learning-speed, it does not necessarily result in a more streamlined agent code. This is intuitive because more learning is deferred to Layer 2, where "passing" behavior is incorporated into the more sophisticated "keep-away" agents.

6 CONCLUSIONS

We have shown that using layered learning, genetic programming can evolve intelligent agents for a cooperative MAS task such as keep-away soccer more quickly, with better fitness. Additionally, layered learning GP allows for a natural decomposition of the MAS learning problem into subproblems, each of which is more easily solved with GP. The keep-away soccer problem is a good test bed for abstracting away the

complexities of simulated soccer and allows for different GP methods to be evaluated and their relative merits compared. It is also easily extended to the full game of robotic soccer, and is highly portable across platforms because our simulator, *TeamBots* [Ba01], *SoccerServer* [An99], and *ECJ* [Lu00] are all written in Java.

Conceptually, we can liken our success with LLGP to the success of human soccer teams. Successful teams are usually made up of players with unique strategies, where learning took place in a bottom-up fashion and individuals first learned to play well together in pairs and small groups, then as a coordinated team. The LLGP-All experiments simulate this kind of behavior, where we attempt to minimize the number of generations needed per layer. Our results indicate that layered learning in GP yields benefits over both standard GP and over hand-coded hierarchical approaches that depend on a large volume of domain knowledge. This is because it is easier and more natural to use the team fitness function to derive an intermediate fitness function, evolve primitive MAS agents, then let the higher-level (Layer 2) GP discover how to compose and refine primitive MAS behavior into complex MAS behavior.

We have considered several extensions to this research. First, developing a full-scale team for the *RoboCup* competition using LLGP would be a good way to test its abilities more thoroughly (however, the focus in this paper was on evaluating MAS task decomposition and improvement of learning accuracy and learning speed). Diversity in populations is also an interesting issue, and our continuing research in LLGP investigates how and whether LLGP promotes diversity. A related question is the degree to which LLGP *reuses* code versus *refining* it in higher layers. Other interesting modifications include developing heterogeneous teams, adding additional lower- and higher-level layers, and hybridizing ADFs and layered learning GP.

Acknowledgments

Support for this research was provided in part by the Army Research Lab under grant ARL-PET-IMT-KSU-07 and by the Office of Naval Research under grants N00014-00-1-0769 and N00014-01-1-0917. We also thank Edmund Burke for providing support for the second author in continuing this work. Finally, thanks to Sean Luke for providing help with ECJ, the GP library used to develop our system.

References

[An99] D. A. Andre. *SoccerServer Manual Ver. 4, Rev. 02*. Available through the World-Wide Web at http://www.robocup.org, 1999.

[As99] M. Asada. Overview of RoboCup-98. In *RoboCup-98: Robot Soccer World Cup II (Lecture Notes in Artificial Intelligence Vol. 1604)*. Springer-Verlag, New York, NY, 1999.

[AT99] D. A. Andre and A. Teller. Evolving Team Darwin United. In *RoboCup-98: Robot Soccer World Cup II (Lecture Notes in Artificial Intelligence Vol. 1604)*. Springer-Verlag, New York, NY, 1999.

[Ba01] T. Balch. *TeamBots* software and documentation. Available through the World-Wide Web at http://www.teambots.org, 2001.

[De90] H. deGaris. Genetic Programming: Building Artificial Nervous Systems Using Genetically Programmed Neural Network Modules". In B. W. Porter *et al*, editors, *Proceedings of the Seventh International Conference on Machine Learning (ICML-90)*, p. 132-139, 1990.

[DC97] M. Dorigo and M. Colombetti. *Robot Shaping: An Experiment in Behavior Engineering.* MIT Press/Bradford Books, 1997.

[Er00] R.I. Eriksson. An initial analysis of the ability of learning to maintain diversity during incremental evolution. In A. A. Freitas, editor, *Data Mining with Evolutionary Algorithms*, p. 120-124. 2000.

[HHC94] I. Harvey, P. Husbands, and D. Cliff. Seeing the light: artificial evolution, real vision. In D. Cliff et al, editors, *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior*. MIT Press/Bradford Books, Boston MA, 1994.

[GH01] S. M. Gustafson and W. H. Hsu. Layered Learning in Genetic Programming for A Cooperative Robot Soccer Problem. In J. F. Miller *et al*, editors, Proceedings of the European Conference on Genetic Programming (EuroGP-2001). Lake Como, Italy. Springer-Verlag, 2001.

[Ka95] C. M. Kadie. *Seer: Maximum Likelihood Regression for Learning-Speed Curves.* Ph.D. Dissertation, University of Illinois at Urbana-Champaign (Technical Report UIUC-DCS-R1874). August, 1995.

[KAK+95] H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, and E. Osawa. RoboCup: The Robot World Cup Initiative. In *Proceedings of the 1995 International Joint Conference on Artificial Intelligence (IJCAI-95)* Workshop on Entertainment and AI/Alife. Montréal, Canada, 1995.

[Ki97] H. Kitano. The RoboCup Synthetic Agent Challenge 97. In Proceedings of the *1997 International Joint Conference on Artificial Intelligence (IJCAI-97)*, Nagoya , Japan, 1997.

[Ko92] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, 1992.

[Ko94] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, MA, 1994.

[Ko98] J. R. Koza. *Genetic Programming III: Darwinian Invention and Problem Solving*. Morgan Kaufmann, Los Altos, CA, 1998.

[LS96] S. Luke and L. Spector. Evolving Teamwork and Coordination with Genetic Programming. In *Genetic Programming 1996: Proceedings of the First Annual Conference*. J. Koza *et al*, eds. p. 141-149. MIT Press, Cambridge, MA, 1996.

[Lu98] S. Luke. Genetic Programming Produced Competitive Soccer Softbot Teams for RoboCup-97. In *Proceedings of the Third Annual Genetic Programming Conference (GP98)*. J. Koza *et al*, eds. p. 204-222. Morgan Kaufmann, Los Altos, CA, 1998.

[Lu00] S. Luke. *Issues in Scaling Genetic Programming: Breeding Strategies, Tree Generation, and Code Bloat*. Ph.D. Dissertation, Department of Computer Science, University of Maryland, College Park, MD, 2000.

[MNH97] H. Matsubara, I. Noda, K. Hiraku. Learning of Cooperative Actions in Multi-agent Systems: A Case Study of Pass Play in Soccer. In *Adaptation, Coevolution, and Learning in Multiagent Systems: Papers from the 1996 American Association for Artificial Intelligence (AAAI) Spring Symposium*, AAAI Technical Report SS-96-01, p. 63-67. AAAI Press, Menlo Park, CA, 1996.

[RB94] J. P. Rosca and D. H. Ballard. Hierarchical Self-Organization in Genetic Programming. In *Proceedings of the Eleventh International Conference on Machine Learning (ICML-94)*, p. 251-258. Morgan Kaufmann, Los Altos, CA, 1994.

[St00] P. Stone. *Layered Learning in Multiagent Systems: A Winning Approach to Robotic Soccer*. MIT Press, Cambridge, MA, 2000.

[SV98] P. Stone and M. Veloso. A Layered Approach to Learning Client Behaviors in the RoboCup Soccer Server. *Applied Artificial Intelligence (AAI) 12(3):165-188*. Taylor and Francis, London, UK, 1998.

[SV00a] P. Stone and M. Veloso. Layered Learning. In *Proceedings of the Eleventh European Conference on Artificial Intelligence (ECAI)*. 2000.

[SV00b] P. Stone and M. Veloso. Multiagent Systems: A Survey from a Machine Learning Perspective. *Autonomous Robots*, 8(3): 345-383. Kluwer Academic Publishers, Norwell, MA, 2000.

[SVR99] P. Stone, M. Veloso, and P. Riley. The CMUnited-98 Champion Simulator Team. In *RoboCup-98: Robot Soccer World Cup II (Lecture Notes in Artificial Intelligence Vol. 1604)*. Springer-Verlag, New York, NY, 1999.

[SX93] M. Schoenauer and S. Xanthakis. Constrained GA Optimization. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms (ICGA-93)*, p. 573-580. Morgan Kaufmann, San Mateo, CA, 1993.

[Ta97] M. Tambe. Towards Flexible Teamwork. *Journal of Artificial Intelligence Research*, 7: 83-124, 1997.

[TAA+99] M. Tambe, J. Adibi, Y. Alonaizon, A. Erdem, G. Kaminka, S. Marsella, and I. Muslea. Building Agent Teams using an Explicit Teamwork Model and Learning, *Artificial Intelligence*, 110:215-240. Elsevier, 1999.

[WM98] J.F. Winkeler and B.S. Manjunath. Incremental Evolution in Genetic Programming. In J.R. Koza, editor, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, p. 403-411. Morgan Kaufmann, San Mateo, CA, 1998.

Adaptive Hierarchical Fair Competition (AHFC) Model for Parallel Evolutionary Algorithms

Jianjun Hu* hujianju@cse.msu.edu

Erik D. Goodman* goodman@egr.msu.edu

Kisung Seo* ksseo@egr.msu.edu

Min Pei* pei@egr.msu.edu

Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824

*Genetic Algorithms Research and Applications Group
Michigan State University
2857 W. Jolly Rd., Okemos, MI 48864

Abstract

The HFC model for parallel evolutionary computation is inspired by the stratified competition often seen in society and biology. Subpopulations are stratified by fitness. Individuals move from low-fitness to higher-fitness subpopulations if and only if they exceed the fitness-based admission threshold of the receiving subpopulation, but not of a higher one. The HFC model implements several critical features of a competent parallel evolutionary computation model, simultaneously and naturally, allowing rapid exploitation while impeding premature convergence. The AHFC model is an adaptive version of HFC, extending it by allowing the admission thresholds of fitness levels to be determined dynamically by the evolution process itself. The effectiveness of the Adaptive HFC model is compared with the HFC model on a genetic programming-based evolutionary synthesis example.

1 INTRODUCTION

Parallel evolutionary algorithms (PEA's) have gained increasing attention in many large-scale application problems including graph-partitioning problems, set partitioning problems, and many commercial efforts in analog circuit synthesis at Analog Design Automation Co. (Liang, 2001), Neolinear Inc (Ochotta, 1996; Krasnicki, 1999) and Genetic Programming Inc. (Andre, 1996). Parallel evolutionary computation models can be largely categorized into three classes (Cantu-Paz, 1998; Nowostawski, 1999): (1) global single- population master-slave models (2) single-population fine grained models, and (3) multi-population coarse-grained (or island) models. As cluster computing and networked PC's have become available in many companies, multi-population parallel models (sometimes combined with master-slave models) have become increasingly popular. Parallel evolutionary algorithms have major advantages over single-population models, including parallel evaluation and rapid exploration with decreased risk of premature convergence. However, current parallel EA's are still not competent vis-a-vis scalability, either with respect to increasing degree of difficulty of the problem or to speedup with an increasing number of processors. It is clear that a competent parallel evolutionary algorithm should have the capability to:

(1) quickly exploit high-fitness individuals as they are discovered. One of these mechanisms is Elitism, which is effective in preserving good individuals, as has been demonstrated in several of the most successful evolutionary multi-objective optimization algorithms, such as NSGAII and SPEAII (Zitzler, 2000).

(2) keep multiple high-fitness individuals simultaneously to facilitate exploration in multiple search areas or directions

(3) maintain diversity of the population to avoid premature convergence

(4) be scalable with respect to increasing number of processors

(5) adapt its parameters for autonomous evolutionary computation.

Multi-population PEA's can be classified into homogeneous models and heterogeneous models. Sprave (1999) proposed a unified model of population structures in PEAs, but his model doesn't concern with the heterogeneity of the sub-populations. In homogeneous parallel EA models, each subpopulation is regarded as playing the same role in evolution. Homogeneous PEA's often lack efficient mechanisms to exploit the newly discovered high-fitness individuals. Although they may keep several high fitness individuals in different demes,

they suffer from the fact that high-fitness individuals may easily dominate all subpopulations by means of the exchange ("migration") process. Heterogeneous parallel EA's are typically more resistant to this phenomenon. For example, the injection island GA (iiGA) (Lin, 1994; Eby, 1999) uses a hierarchical structure, typically stratifying subpopulations according to the level of resolution of the representation, allowing control of the tradeoff between low-resolution exploration and high-resolution exploitation. The iiGA has also been used with different fitness functions in various subpopulations, even if they used the same problem representation. Aickelin (1999) also proposed such a PEA, which he called a pyramidal EA, in which the hierarchical structure of the subpopulations is defined by a hierarchy of fitness functions.

In a recent paper (Hu and Goodman, 2002), we proposed the Hierarchical Fair Competition (HFC) model for parallel evolutionary computation. The HFC model is inspired by the observation of a strategy employed in some societal and biological systems to maintain different high-fitness individuals in a whole population. HFC turns out to have the features of a competent PEA cited above except the adaptability of (6). In this paper, we introduce an adaptive version of the HFC model, in which the admission thresholds are automatically determined and adjusted in the evolutionary process. In Section 2, the metaphor and the HFC model are described relative to the above features. In Section 3, an adaptive mechanism for determining the parameters of the HFC model is presented, along with the algorithm. We apply the AHFC model to a genetic programming problem and compare it with the static HFC model in Section 4. The conclusions and discussion are provided in Section 5.

2 THE HIERARCHICAL FAIR COMPETION MODEL (HFC) FOR PARALLEL EVOLUTION

2.1 MOTIVATION AND BACKGROUND OF HFC

The HFC model originates from an effort to combat the premature convergence phenomenon in traditional genetic algorithms and genetic programming. In a traditional GA, as the evolutionary process goes on, the average fitness of the population gets higher and higher, so that new individuals tend to survive only if they have similarly high fitness. New "explorer" individuals in fairly different regions of the search space usually have low fitness, until some local exploration and exploitation of their beneficial characteristics has occurred. So a standard EA tends to concentrate more and more of its search effort near one or more early-discovered peaks, and to get "stuck" near these attractors (or local optima). It is clear that in a standard EA, there exists a severely unfair competition. That is, selection pressure makes high-fitness individuals reproduce quickly and thus supplant other individuals with lower fitness, some of which may lie in the vicinity of the global optimum, when if their neighborhood were explored more thoroughly, much higher-fitness individuals would be found. This fact holds true even when we find search points near a global optimum, as long as they are not close enough to have high fitness relative to those near other, earlier-explored local optima. This "unfair" competition contributes a lot to the slow search progress of many EA's when confronted with difficult, high- dimensionality, multi-modal problems. To address this unfair competition problem, we need allow young but promising individuals (*i.e.,* those in relatively newly-found regions, which may ultimately give rise to high-fitness offspring, but which are currently not of high fitness) to "grow up" and, at an appropriate time, join in the cruel competition process and be kept for further exploitation or be killed (as appropriate) when they are demonstrated with some confidence to be bad. At the same time, we hope to maintain the already-discovered high-fitness individuals and select from them even more promising individuals for exploitation without killing younger individuals. Following the tradition of getting inspiration from biology, we find that in some societal and biological systems, there exists an efficient mechanism that can maintain *and foster* potentially-high-fitness individuals (or, more accurately, potential progenitors of high-fitness individuals) efficiently. This is the hierarchical fair competition (HFC) principle as discussed below.

2.2 THE METAPHOR OF HFC: HIERARCHICAL FAIR COMPETITION IN SOCIETAL AND BIOLOGICAL SYSTEMS

Competition is widespread in societal and biological systems, but diversity remains large. After close examination, we find there is a fundamental principle underlying many types of competition in both societal and biological systems: the Fair Competition Principle.

2.2.1 The Fair Competition Principle in Societal Systems

In human society, competitions are often organized into a hierarchy of levels. None of them will allow unfair competition – for example, a young child will not normally compete with college students in a math competition. We use the educational system to illustrate this principle in more detail.

In the education system of China and many other developing countries, primary school students compete to get admission to middle schools and middle school students compete for spots in high schools. High school students compete to go to college and college students compete to go to graduate school (Fig. 1) (in many Western countries, this competition starts at a later level, but is eventually present, nonetheless). In this hierarchically structured competition, at each level, only individuals of roughly equivalent ability will participate

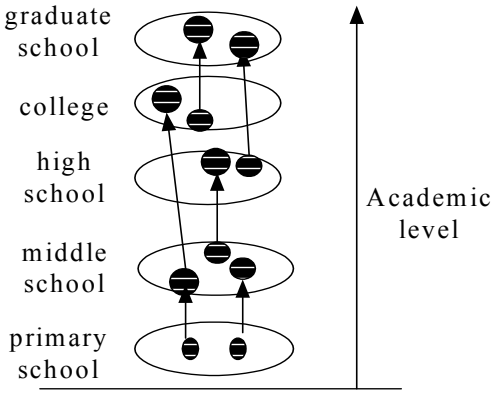


Figure 1: In education systems, low level students compete to get admission to higher level schools.

in any competition; *i.e.,* in such societal systems, only fair competition is allowed. This hierarchical competition system is an efficient mechanism to protect young, potentially promising individuals from unfair competition, by allowing them to survive, learn, and grow before joining more intense levels of competition. Individuals that "lose" in these fair competitions were selected against while competing fairly only against their peers. Students compete fairly against others in their grade level because they are usually of similar absolute fitness levels, having been exposed to similar amounts of education and experience.

An interesting phenomenon sometimes found in societal competitions is the "child prodigy." A ten-year-old child may have some extraordinary academic ability. These prodigies may skip across several educational levels and begin to take college classes at a young age. An individual with sufficient ability (fitness) is allowed to join any level of competition. This also suggests that in subpopulation migration, we should migrate individuals according to their fitness levels, rather than according to "time in grade."

With such a fair competition mechanism that exports high-fitness individuals to higher-level competitions, societal systems reduce the prevalence of unfair competition and the unhealthy dominance or disruption that might otherwise be caused by "early-achieving" individuals.

2.2.2 The Fair Competition Principle in Biological Systems

It is somewhat surprising that in "cruel" biological/ ecological systems, the fair competition principle also holds in many cases. For example, there are mechanisms that reduce unmatched or unfair competition between young animals and mature ones. Among mammals, young individuals often compete with their siblings under the supervision of parents, but not directly with other mature individuals, since their parents protect them against other adults. When the young grow up enough, they leave their parents and join the competition with other mature individuals. Evolution has found the mechanisms of parental care and sibling competition to be useful in protecting the young and allowing them to grow up and develop their full potentials. Fair competition seems to be beneficial to the evolution of many species.

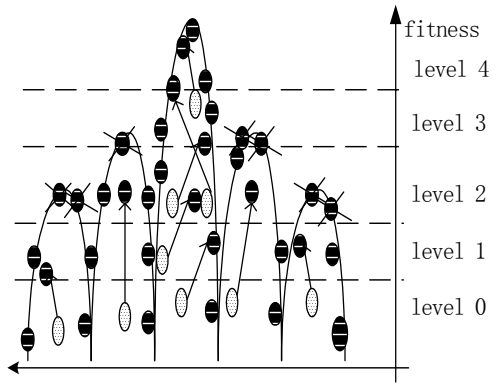


Figure 2: HFC model extends the search horizontally in search space and vertically in fitness dimension and kills bad individuals at appropriate times while allowing promising young individuals grow up continuously

2.3 THE HFC MODEL

Inspired by the fair competition principle and the hierarchical organization of competition within subpopulations in societal systems, we propose the Hierarchical Fair Competition parallel model (HFC), for genetic algorithms, genetic programming, and other forms of evolutionary computation.

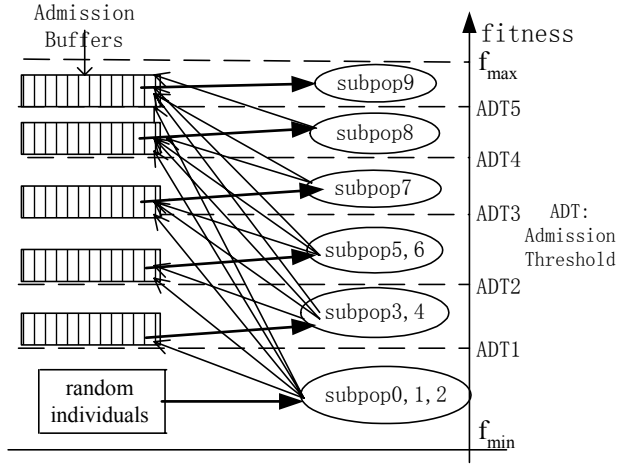


Figure 3: In HFC model, subpopulations are organized in a hierarchy with ascending fitness levels. Each level (with one or more subpopulations) accomodates individuals within a certain fitness range determined by the admission thresholds

In this model (Fig 3), multiple subpopulations are organized in a hierarchy, in which each subpopulation can only accommodate individuals within a specified range of fitness. The entire range of possible fitnesses is spanned by the union of the subpopulations' ranges. Conceptually, each subpopulation has an admission buffer that has an ***admission threshold*** determined either initially (fixed) or adaptively. The admission buffer is used to collect qualified candidates, synchronously or asynchronously, from other subpopulations. Each subpopulation also has an ***export threshold*** (fitness level), defined by the admission threshold of the next higher-level subpopulation. Only individuals whose fitnesses are between the subpopulation's admission threshold and export threshold are allowed to stay in that subpopulation. Otherwise, they are exported to the appropriate higher-level subpopulation. Exchange of individuals is allowed only in one direction, from lower-fitness subpopulations to higher-fitness subpopulations, but migration is not confined to only the immediately higher level.

Each subpopulation can have the same or different sizes, operators, and other parameters. However, considering that there are often more low-fitness peaks than high-fitness peaks, we tend to allocate larger population sizes or more subpopulations to lower fitness levels, to provide extensive exploration; and we tend to use higher selection pressures in higher-fitness-level subpopulations to ensure efficient exploitation. As it is often easier to make a big fitness jump in a lower level subpopulation, we often end up using larger fitness ranges for low-level subpopulations, and smaller ranges for high-level subpopulations (Fig. 2), but, of course, that depends on the properties of the fitness landscape being explored. The critical point is that the whole range of possible fitnesses must be spanned by the union of the ranges of all levels of subpopulations. Of course, the highest-level subpopulation(s) need no export threshold (unbounded above) and the lowest-level subpopulation(s) need no admission threshold (unbounded below).

Exchange of individuals can be conducted synchronously after a certain interval, or asynchronously, as in many parallel models. At each moment of exchange, each individual in each subpopulation is examined, and if it is outside the fitness range for its subpopulation, it is exported to the admission buffer of a subpopulation with an appropriate fitness range. When a new candidate is inserted into an admission buffer, it can be inserted into a random position or inserted by sorting (or a null buffer may be used, inserting migrants directly into the receiving subpopulation, using some replacement rule). After export, each subpopulation imports the appropriate number of qualified candidates from its admission buffer into its pool. Subpopulations (especially at the base level) fill any spaces still open after emptying their admission buffers by generating new individuals at random to fill the spaces left by the exported individuals.

The number of levels in the hierarchy or number of subpopulations (if each level has only one subpopulation) can be determined initially or adaptively. In the static HFC model, we must manually decide into how many levels the fitness range will be divided, the fitness thresholds, and all other GA parameters. In a dynamic HFC model, we can dynamically change the number of levels, number of subpopulations, size of each subpopulation, and admission and export fitness thresholds. As will be seen below, a benefit of the adaptive HFC model (an example of a dynamic HFC) is that it can adaptively allocate search effort according to the characteristics of the search space of the problem to be solved, thereby searching more efficiently (initial research on various methods for adaptation of thresholds is in preparation for reporting elsewhere). However, even "coarse" setting of the parameters in a static HFC model has yielded major improvement in search efficiency over current EA's on example problems.

Another useful extension to HFC used here is to introduce one or more *sliding* subpopulations, with dynamic admission thresholds that are continually reset to the admission threshold of the level in which the current best individual has been found. Thus, these subpopulations provide additional search in the vicinity of the advancing frontier in the hierarchy.

2.3.1 HFC as a competent parallel model for parallel evolutionary computation

(1) While low-fitness individuals can persist long enough to allow thorough exploration, as soon as they produce high-fitness offspring, the offspring can advance to higher-fitness levels immediately for further exploitation, to compete and be recombined with other high-fitness individuals.

(2) The HFC model maintains a large number of high-fitness individuals in high-fitness-level subpopulations without threatening lower-fitness (but perhaps promising) individuals. Thus possibly promising new search locales can persist long enough to be appropriately exploited.

(3) HFC provides another mechanism for maintaining diversity. First, the diversity of the population is ensured by the stratification in the fitness space. Second, continuous introduction of random individuals into the lowest-level subpopulations and the promotion of their high-fitness offspring to upper-level subpopulations can be regarded as the introduction of entropy and randomness into the overall evolutionary system. Actually, looking from low-fitness levels to higher-fitness levels, we observe increasing order in the population. The HFC evolution is thus a self-organizing process in which the highest order is achieved at the top fitness level. This mechanism reduces the chance of HFC becoming "stuck" at local optima and helps it explore new search areas. HFC thus implements implicitly a

multi-start or re-initialization mechanism on a continual basis.

(4) The HFC model quickly captures superior offspring and moves them to a place where they are free to compete with, and be recombined with, each other. This produces an effect similar to the elitism often used in multi-objective evolutionary computation, such as NSGAII or SPEAII (Zitzler *et al.,* 2000), in which superior individuals are also kept separately. At that level, we can control the intensity of selection to determine the tradeoff between exploitation of those high-fitness individuals and exploration in their neighborhoods.

(5) HFC has a good scalability to more processing hosts. As more processors are available, they can be distributed to different fitness levels – either to low-level subpopulations for more extensive exploration, or to the higher-level ones for intensive exploitation of high-fitness individuals.

One of the major difficulties in the HFC evolutionary algorithm is the determination of the admission thresholds for a given problem. As the fitness landscape is often unknown before evolutionary search, it is hard to define these admission thresholds initially. Considering that admission thresholds in HFC are only used to segregate the whole population to avoid unfair competition, the behavior of the search is generally not extremely sensitive to the values of these admission thresholds, so that it is not necessary to set them to exactly optimal values. The only requirement for these thresholds is that the union of the fitness level ranges (which is determined by these admission thresholds) span the entire range of possible fitnesses. Based on this analysis, we propose an automatic admission thresholds determination mechanism for HFC model.

3 THE ADAPTIVE HFC MODEL

In the static HFC model, we need to determine the number of subpopulations, the number of fitness levels, the relationship of subpopulations to fitness levels and the admission thresholds of each fitness level. All the admission thresholds are determined based on some initial exploration of the fitness landscape of the problem, such as the range of the fitness or distribution of early-discovered peaks. The threshold adaptation mechanism proposed here enables us to be relieved from this prerequisite expertise in the problem space. All we must decide is the number of admission levels (N_l).

Since in HFC, random individuals are continuously inserted into subpopulations of the base fitness level, the export threshold of the base fitness level can be set as the average fitness of the whole population after several (*nCalibGen*) generations. In AHFC, this is called the **calibration stage**, which determines the level of the fitness value of frequently encountered ("normal") individuals with respect to random individuals. So the base level is used to export normal individuals to higher levels for further exploitation. At the end of the calibration process, the standard deviation σ_f and the max fitness $f_{\max}$ of individuals at the highest level, the average fitness f_μ of individuals at the base level are calculated. Then the fitness range of each level can be calculated by the following formula:

Admission threshold of base level= - ∞ (1)

Admission threshold of the first level = f_μ (2)

Admission threshold of the highest fitness level =

$$f_{\max} - \sigma_f \quad (3)$$

Admission thresholds of other fitness levels L_i, are determined by:

$$f_\mu + L_i \times (f_{\max} - \sigma_f - f_\mu)/(N_l - 2) \qquad i = 1,\ldots, N_l - 1 \quad (4)$$

Table 1: Adaptive Heterogeneous HFC Algorithm for Parallel EA's

1. Initialization
   Determine $\vec{P}_{EA}$: parameters for standard multi-population EA. (We assume here using one set of parameters for all subpopulations)
   N_l :Number of levels of the hierarchy
   nCalibGen: calibration generations
   nUpdateGen: admission threshold update interval
   nExch: generations between admission process exchanges
   gen = 1: current generation
2. Do
   if *gen* < *nCalibGen* (in calibration stage)
      run EA without exchange
   else if *gen* = *nCalibGen* (calibration stage ends)
      determine the admission thresholds for each level by formulas (1) - (4)
   else if *gen*% *nExch* = 0
      Do for each subpopulation from lowest level to highest level {
         Examine fitness of each individual and export to corresponding subpopulation at higher level for which fitness range accommodates this exported individual (replacing worst individual in target subpopulation)}
      end do
   else if *gen* % *nUpdateGen* =0
   update admission thresholds of all but the base level by (3), (4)
   gen ++

until the stopping criterion is satisfied.
   return the highest-fitness individual(s) from the highest-level subpopulation
End

However, it is clear that as the evolutionary search goes on, higher-fitness individuals are continuously discovered that ruin the segregation by the above admission thresholds determined at the initial calibration stage. So a dynamic **admission threshold updating** mechanism is proposed here. After each *nUpdateGen* generations, the maximal fitness, $f_{\max}$, and the fitness standard deviation of the top level sub-populations, σ_f, are recomputed to determine the admission threshold of all the fitness levels except the base level and the first level, by (3) - (4).

To enable efficient search, the mapping relationship of sub-populations to all levels also needs to be adapted dynamically. It is obvious that at initial stage, as all individuals are randomly generated, these individuals usually have low fitness. So most of the subpopulations should belong to the base level. As higher-level individuals discovered, more subpopulations should be allocated to higher levels to exploit high-fitness individuals. The following scheme is used in this paper: firstly, all subpopulations are allocated to base level. After the calibration stage, subpopulations are then evenly allocated to each level. Extra subpopulations can be allocated to higher levels (if aggressive exploitation is desired) or to lower level (if intensive exploration is desired).

This AHFC algorithm works like a string. At the initial stage, it is quite compressed, but gradually, the string stretches to accommodate individuals with a larger range of fitness. The whole algorithm of AHFC is given in Table 1. For simplicity, we give the pseudo code only for the adaptive HFC model with synchronous exchanges (no buffers).

4 EXPERIMENTS

The adaptive HFC model for Genetic Programming (HFC-GP) has been applied to a real-world analog circuit synthesis problem that was first pursued using GP with a static HFC (Hu, 2002). In this problem, an analog circuit is represented by a bond graph model (Seo, 2001; Fan, 2001) and is composed of inductors (I), resistors (R), capacitors (C), transformers (TF), gyrators (GY), and Sources of Effort (SE). Our task is to synthesize a circuit, including its topology and sizing of components, to achieve specified behavior. The objective is to evolve an analog circuit with response properties characterized by a pre-specified set of eigenvalues. By increasing the number of eigenvalues specified, we can define a series of synthesis problems of increasing difficulty, in which premature convergence problems become more and more significant when traditional GP methods are used.

Circuit synthesis by GP is a well-studied problem that generally demands large computational power to achieve good results. Since both topology and the parameters of a circuit affect its performance, it is easy to get stuck in the evolution process.

4.1.1 Experiments on an Analog Circuit Synthesis Problem

Four circuits with increasing difficulty are to be synthesized, with eigenvalue sets as specified in Table 2. Circuits were evolved with single-population GP, multiple-population GP, HFC-GP, and AHFC-GP. The GP parameter for the single-population GP is shown in cell (1,2) of Table 3. The GP parameters for the multi-population GP were the same as for the single-population GP, except that the total population is divided into subpopulations with sizes shown in cell (2, 2) of Table 3. A one-way ring migration topology was used.

The parameters for the HFC-GP were the same as for the multi-population GP, except that the ring migration was replaced by the HFC scheme. The fitness admission thresholds were set based on our prior experience with such eigenvalue problems. In this problem, we defined a fitness admission threshold for each subpopulation (one subpopulation per level, in this case) as shown in cell (2, 3) of Table 3. Subpopulation 15 was used as a "sliding" subpopulation to aggressively explore the fitness frontier.

Table 2: Target Eigenvalues

Problem 1: 6-eigenvalue problem
$-2 \pm 3.3i,\ -7.5 \pm 4.5i,\ -3.5 \pm 12.0i$
Problem 2: 8-eigenvalue problem
$-2 \pm 3.3i,\ -7.5 \pm 4.5i,\ -3.5 \pm 12.0i,\ -3.4 \pm 12.0i$
Problem 3: 10-eigenvalue problem
$-2 \pm 3.3i,\ -7.5 \pm 4.5i,\ -3.5 \pm 12.0i,\ -3.4 \pm 12.0i,\ -10.0 \pm 8.0i$
Problem 4: 12-eigenvalue problem
$-2 \pm 3.3i,\ -7.5 \pm 4.5i,\ -3.5 \pm 12.0i,$
$-3.4 \pm 12.0i,\ -10.0 \pm 8.0i,\ -1.5 \pm 3.0i$

Table 3: Parameter Settings for GP

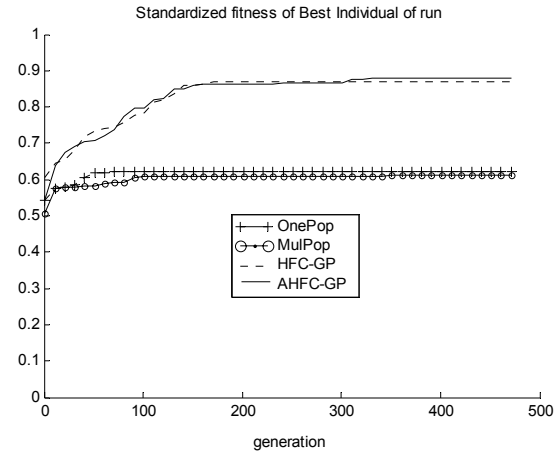
Parameters of Single Population GP	Popsize: 2000 init.method = half_and_half init.depth = 3-6 max_nodes = 800 max_depth = 13 crossover rate = 0.9 mutation rate = 0.1 max_generation = 1000
Additional Parameters of Multi-Population GP	Number of subpopulations = 15; Size of subpop 2 to 14 = 100 size of subpop 1 = 300 size of subpop 15 = 400 migration interval = 10 generations migration strategy: migrate (copy) 10 best individuals to the next subpopulation in the ring to replace its 10 worst individuals
Additional Parameters of HFC-GP	admission_fitnesses of: subpop 1 = -100000.0 subpop 2 to 14: 0.65, 0.68, 0.72, 0.75, 0.78, 0.80, 0.83, 0.85, 0.87, 0.9, 0.92, 0.95 subpop 15 = varying
Additional Parameters of AHFC-GP	*nUpdateGen*= *nCalibGen* = 10 *nExch* = 10 N_l =8

The parameters of AHFC-GP were nearly identical to those of the HFC, except that we don't need to determine the admission thresholds of each level.

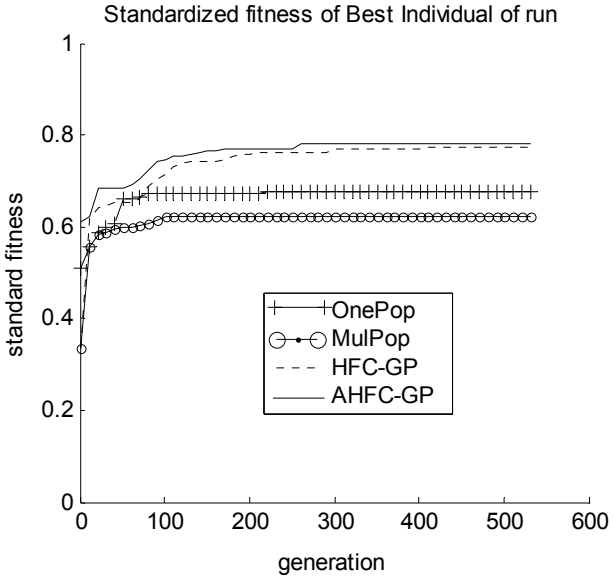
The performances of the four approaches were assessed on four problems with increasing difficulty. Each experiment was run ten times, with the average of the results reported in Fig.4, where the four GP methods are indicated by

OnePop: Single-population GP
MulPop: multi-population GP (ring topology)
HFC-GP: HFC model for GP
AHFC-GP: Adaptive HFC model for GP

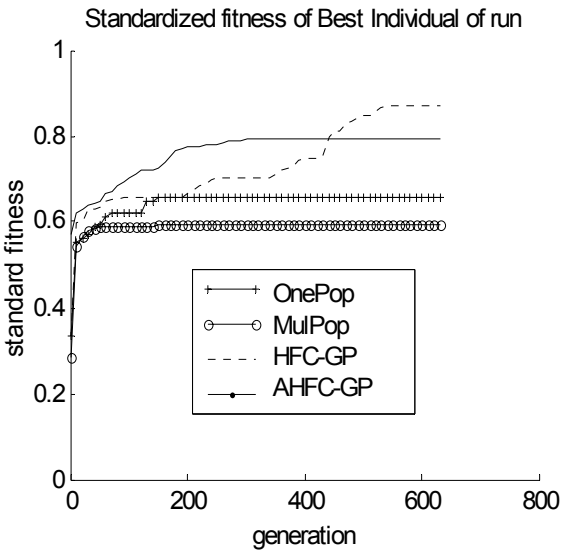
From Figure 4, it is impressive to see that in all four problems, both AHFC and HFC performed dramatically better than the other algorithms vis-à-vis best of run, and the improvement was more dramatic on the more difficult



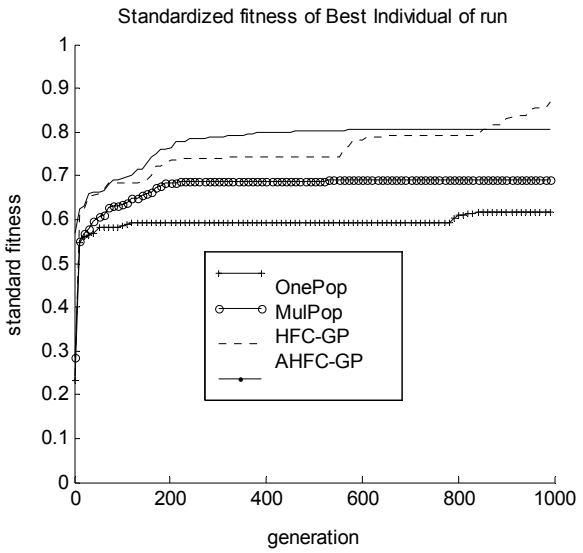
a) 6-eigenvalue problem



b) 8-eigenvalue problem



c) 10-eigenvalue problem



d) 12-eigenvalue problem

Figure 4. Fitness of Best Individual to Date vs. Generation: dashdot (OnePop), solid (MulPop), dashed (HFC)

problems. The superior performance at the initial generations may have resulted from the rapid exploitation of superior individuals, in a single subpopulation, in comparison to the ring parallel GA. Yet convergence in the HFC and AHFC was much slower than in the single- and multi-population GP runs. In fact, we observe relatively steady improvement during the runs for this set of problems. For the easier problems, the (dynamic) AHFC actually out-performed the (static) HFC slightly, in spite of the rich experience on this class of problems that was used for setting the HFC thresholds. The fact that the HFC ultimately surpassed the AHFC on the two harder problems indicates that there is room for improvement of the AHFC scheme used here. However, the fact that it is competitive with human-determined

static values based on prior experience shows that it is a step in a beneficial direction.

5 CONCLUSIONS AND FUTURE WORK

Based on our analysis of the role of admission thresholds used in the HFC model and our experiments on a series of difficult, highly epistatic real-world problems, it has been demonstrated that the adaptive HFC model can work nearly as well as the original HFC model, and even better in some cases, without any prerequisite knowledge of the fitness landscape of the problem. The dynamic allocation of the subpopulations to fitness levels also improves the search efficiency. These adaptation mechanisms make our algorithm to be easily plugged into new problems without much parameter tuning. Our experiments demonstrated the effectiveness of the HFC and AHFC models in improving significantly both the search speed and the quality of the best solutions found compared with standard EAs.

This paper represents a first step toward autonomous parallel evolutionary computation based on the HFC model. The second step is the automation of the adaptive distribution of the computing resource among levels. We expect that the number of subpopulations, the number of fitness levels, the distribution of subpopulations to each level, along with the admission thresholds, can all be determined adaptively, in which case we would have an autonomous parallel evolutionary computation model in which the communication topology and migration scheme are all decided by the evolutionary process itself, according to the characteristics of the problem at hand.

In this paper, we implemented the synchronous version of the AHFC model and simulated parallel genetic programming on a single PC. More consideration about the communication cost and asynchronous adaptation mechanism of the AHFC model in the case of a large population is needed. The scalability of the AHFC model with respect to more processors also needs to be proved with experiments on real parallel cluster computing facilities, which is on the top of our task list.

Acknowledgment

The authors acknowledge the assistance of Prof. Ronald C. Rosenberg and Zhun. Fan in defining and exploring the example problem presented here. This work was supported by the National Science Foundation under contract DMI 0084934.

References

J.M. Liang; T. McConaghy; A. Kochlan; T. Pham; G. Hertz. “Intelligent Systems for Analog Circuit Design Automation: A Survey,” *Proceedings World Multiconference on Systemics, Cybernetics and Informatics,* Vol. IX. SCI 2001/ ISAS 2001, Orlando, Florida (USA), 2001.

E. S. Ochotta, R. A. Rutenbar, L. R. Carley, “Synthesis of High-Performance Analog Circuits in ASTRX/ OBLX,” *IEEE Trans. CAD*, Vol. 15, pp. 273-294, Mar., 1996.

M. Krasnicki, R. Phelps, R. Rutenbar, and R. Carley. “Maelstrom: Efficient simulation-based synthesis for custom analog cells,” *Proceedings of the 1999 ACM/IEEE Design Automation Conference*, 1999.

D. Andre and J. R. Koza. “Parallel genetic programming on a network of transputers,” in Angeline, Peter J. and Kinnear, Kenneth E., Jr. (eds.), *Advances in Genetic Programming,* MIT Press, Cambridge, MA, 1996.

E. Cantu-Paz, “A survey of parallel genetic algorithms,” *Calculateurs Parallels*, 10(2), Paris, Hermes. 1998.

J. Sprave. A unified model of non-panmictic population structures in evolutionary algorithms. In P. J. Angeline and V. W. Porto, editors, *Proc. 1999 Congress on Evolutionary Computation (CEC'99)*, vol 2, p. 1384-1391, Washington D.C., 1999. IEEE Press, Piscataway NJ.

N. Mariusz. and R. Poli, “Review and Taxonomy of Parallel Genetic Algorithms,” Technical Report, University of Birmingham, School of Computer Science, No. CSRP-99-11, May 1999.

S.C. Lin, E. Goodman, and W. Punch, “Coarse-Grain Parallel Genetic Algorithms: Categorization and New Approach,” *IEEE Conf. on Parallel and Distrib. Processing*, Nov., 1994.

D. Eby, R. C. Averill, E. Goodman, and W. Punch, “Optimal Design of Flywheels Using an Injection Island Genetic Algorithm,” *Artificial Intelligence in Engineering Design, Analysis and Manufacturing*, 13, p. 389-402, 1999.

U. Aickelin, “A Pyramidal Evolutionary Algorithm with Different Inter-Agent Partnering Strategies for Scheduling Problems,” *Genetic and Evolutionary Computation Conference Late-Breaking Papers*, E. Goodman, ed., ISGEC Press, San Francisco, pp. 1-8. 2001.

J.J. Hu, E.D. Goodman, “The Hierarchical Fair Competition (HFC) Model for Parallel Evolutionary Algorithms,” *Proceedings of the 2002 Congress on Evolutionary Computation: CEC2002*, (forthcoming), IEEE, Honolulu, Hawaii, 2002.

K. Seo, E. Goodman, and R. Rosenberg, “First Steps toward Automated Design of Mechatronic Systems Using Bond Graphs and Genetic Programming,” *Proc. Genetic and Evolutionary Computation Conf. - 2001*, July 7-11, Morgan Kaufmann, San Francisco, p. 189, 2001.

Z. Fan, J.J. Hu, K. Seo, E. Goodman, R. Rosenberg, and B. Zhang, “Bond Graph Representation and GP for Automated Analog Filter Design,” *Genetic and Evolutionary Computation Conference Late Breaking Papers*, pp. 81-86, 2001.

E. Zitzler, K. Deb, and L. Thiele, “Comparison of Multiobjective Evolution Algorithms: Empirical Results,” *Evolutionary Computation*, 8(2), pp. 173-195, 2000.

Structure Fitness Sharing (SFS) for Evolutionary Design by Genetic Programming

Jianjun Hu[+], Kisung Seo[+], Shaobo Li*, Zhun Fan[+], Ronald C. Rosenberg[Ψ], Erik D. Goodman[+]

hujianju@cse.msu.edu
[+]Genetic Algorithms Research and Applications Group (GARAGe)
Michigan State University
East Lansing, MI, 48824

*CAD Institute
Guizhou University of Technology
Guiyang, 550003, China

[Ψ]Department of Mechanical Engineering
Michigan State University
East Lansing, MI, 48824

Abstract

Balanced structure and parameter search is critical to evolutionary design with genetic programming (GP). Structure Fitness Sharing (SFS), based on a structure labeling technique, is proposed to maintain the structural diversity of a population and combat premature convergence of structures. SFS achieves balanced structure and parameter search by applying fitness sharing to each unique structure in a population, to prevent takeover by the best structure and thereby maintain the diversity of both structures and parameters simultaneously. SFS does not require definition of a distance metric among structures, and is thus more universal and efficient than other fitness sharing methods for GP. The effectiveness of SFS is demonstrated on a real-world bond-graph-based analog circuit synthesis problem.

1 INTRODUCTION

Genetic programming has been applied successfully to a rich variety of problems such as machine code evolution (Nordin, 1997), quantum algorithm design (Spector, 1999), cellular automaton rule discovery, and soccer-playing program evolution (Andre 1999). GP has been particularly effectively used as an efficient Darwinian Invention Machine that enabled Koza *et al.* to achieve human-competitive results in analog circuit design and in the transmembrane segment identification problem (Koza 1999). Indeed, one of GP's most significant features is the ability to simultaneously evolve both a structure and its parameters, opening up promising applications in many real-world engineering design problems and in neural network design. In all of these problems, the objective is to search for an open-ended structure, together with its related parameters, to achieve several desired goals. Genetic programming – especially evolutionary design by genetic programming – is recognized as making a high demand on computational resources (Koza, 1999). To some extent, this demand can be traced to the premature convergence problem, especially convergence of the structures in a GP population; it can be ameliorated using diversity-maintenance techniques for the population. Based on an analysis of the weak causality of GP, the new concept of Structure Fitness Sharing (SFS), based on a structure labeling technique, is proposed to achieve balanced structure and parameter search by maintaining the diversity of both structures and parameters at all times. This method does not require definition of a distance metric, and is thus very efficient compared to other fitness sharing methods. Its effectiveness is demonstrated on a real-world bond-graph-based analog circuit synthesis problem using GP.

2 THE DIVERSITY PROBLEM IN EVOLUTIONARY DESIGN BY GP

2.1 CATEGORIES OF EVOLUTIONARY DESIGN PROBLEMS

Most evolutionary design problems can be classified into one of three types:

TYPE I: Fixed structure with fixed number of parameters.

These problems are essentially parameter optimization problems – the task is to optimize the parameters of a given structure. Genetic algorithms, simulated annealing, evolutionary programming, evolution strategies, and even gradient-based optimization techniques are often used here.

TYPE II: Variable structure with no parameters.

This type includes problems such as algorithm design, program induction and logic design, in which only structure search is needed. These problems are well suited for GP, which intrinsically manipulates the program structure, often represented as a tree. Of course, some of these problems can be solved with

genetic algorithms, simulated annealing, and other techniques, by using a somewhat indirect representation of the structure.

TYPE III: Variable structure with variable number of parameters (of course, there can also be variable structure with a fixed number of parameters, but that is not the typical case – number of parameters usually varies with the size of the structure).

Many of the most interesting evolutionary design problems belong to this third category, in which a structure is sought within a topologically open-ended space, but the fitness of a structure can often only be evaluated after parameters are assigned to key variables associated with the structures evolved. Since the structure is varied during the search process, the number of parameters and their semantics change frequently. Such problems include analog circuit design (Koza, 1999), mechanical system design (Fonseca, 1993), and neural network design (Oliker, 1992). Although a GA with a variable-length representation can be used here, GP, with its outstanding capability to search simultaneously for a good structure and for appropriate parameters, distinguishes itself as the most important tool for this kind of open-ended design problem.

2.2 PREMATURE CONVERGENCE AND DIVERSITY IN THREE TYPES OF EVOLUTONARY DESIGN PROBLEMS

TYPE I problems are often described as parameter optimization problems, readily addressable by GA. Premature convergence in GA has been well studied. Common diversity maintenance techniques include crowding (DeJong, 1975), deterministic crowding (Mahfoud, 1992), and fitness sharing (Goldberg, 1989). The fitness derating method (Beasley, 1993), a multi-objective method, employs fitness sharing in a popular and effective way.

The premature convergence problem when GP is applied to TYPE II problems has also been well studied. Most of the resulting methods are derived from GA, but with some specific consideration of the GP context. In multi-objective genetic programming, Rodriguez (Rodriguez-Vazquez, 1997) uses the MOGA approach with fitness sharing being performed in the fitness space, and extends it to genetic programming. Though easier to implement, it remains an open question whether diversity of fitness values is generally a true indicator of the diversity of a population – a measure which should actually be based on the parameter space. DeJong *et al.* (DeJong, 2001) use the multi-objective method to explicitly promote diversity by adding a diversity objective. In their method, a distance measure defined as follows is used in the diversity objective. The distance between two corresponding nodes is zero if they are identical and one if they are not. The distance between two trees is the sum of the distances of the corresponding nodes – i.e., distances between nodes that overlap when the two trees are overlaid, starting from the root. The distance between two trees is normalized by dividing by the size of the smaller of the two trees. The diversity objective is defined as the average squared distance to other members of the population. An improved version of the above distance metric between two trees is proposed in Ekart and Nemeth (2000) and used to do fitness sharing in GP. Their method includes the following three steps:

1) The two GP trees to be compared are brought to the same tree-structure (only the contents of the nodes remain different).
2) The distance between each pair of symbols situated at the same position in the two trees is computed.
3) The distances computed in the previous step are combined in a weighted sum to form the distance of the two trees.

The major improvement of this method is that it differentiates the types of nodes when calculating the distance between two nodes. It first divides the GP functions and terminals into several subsets. For nodes with types belonging to the same subset, it calculates the relative distance. For nodes with types belonging to different subsets, it uses a defined function to make sure that the distance between nodes from different subsets is larger than that between nodes of the same subset. It also considers the fact that a difference at some node closer to the root could be more significant than a difference at some node farther from the root, using a multiplier K to distinguish them. Edit distance and phenotypic distance for fitness sharing for GP are also tested in their experiment. The former gets slightly better accuracy but with relatively high computational cost. The latter doesn't provide much improvement over the original GP without fitness sharing.

Implicit fitness sharing (McKay, 2000) has also been applied to GP. Instead of calculating the distance between the structures of GP trees, it is a kind of phenotypic (behavior-based) fitness sharing method. The fitness is "shared" based on the number of other individuals who have similar behaviors, capabilities or functions. Implicit fitness sharing provides selection pressure for each individual to make different predictions from those made by other individuals.

Population diversity of TYPE III problems in GP has not been investigated thoroughly. These problems are characterized by the need for simultaneous optimization of topology and parameters. In a GP population, structure diversity is needed to enable efficient topology exploration, which is the main objective, in most case, for discovery of innovative designs. At the same time, the goodness (or fitness) of a structure can only be evaluated after sufficient parameter exploration within the same structure. Thus, the parameter diversity of each structure also needs to be maintained. As a result, in the context of variable structure and parameter design by GP, the

population diversity has some significant differences from that of a GA, in the following respects:

- Number of peaks

  When applying fitness sharing in GA, two assumptions are made. One is that the number of peaks is known or can be estimated. The second is that the peaks are almost evenly distributed. In many problems of GA, a relatively limited number of peaks is expected to enable efficient use of fitness sharing. However, in TYPE III problems, each structure may have a huge number of peaks with respect to its parameter space, while in the structure space, each structure is a distinct peak, since the structure space is not a continuous space, but rather a highly nonlinear discrete space.

- Continuity of search space

  In GA, many problems can be considered as defined in an approximately continuous space, although sometimes certain aspects have distinctly discrete behavior. However, in TYPE III problems, GP deals with a highly discrete structure space that also has a huge continuous space (of parameter values), since for each structure, the search for appropriate parameters can be regarded as an instance of GA search.

- Constraints

  In GA, only parameter constraints exist. However, in TYPE III problems, GP must deal with both structure constraints and parameter constraints.

The demand for structure diversity as well as parameter diversity makes the existing fitness sharing methods inefficient for Type III problems. For fitness-space-based fitness sharing (Rodriguez-Vazquez, 1997) and the implicit fitness sharing (McKay, 2000) methods, significant parameter diversity is lost since they do not promote coexistence of individuals with the same structure but with different parameters in order to enable efficient parameter search. Fitness sharing with the distance metric, as in (Ekart, 2000; KeJong, 2001), is also inefficient in this case. First, the computational cost is still demanding, since in TYPE III problems, a complex structure and its parameters often require a big tree – perhaps 1000 - 2000 nodes in most of our experiments – especially when parameters are normally represented by a numeric subtree such as Koza uses (Koza, 1999). Second, but more importantly, the underlying assumption of the above distance metrics is that structural dissimilarity measured between two GP trees meaningfully reflects the dissimilarity in function between the two structures. However, as the structure space represented by a GP tree is a highly non-linear space, in most cases, a change of a single (non-parameter) node changes the behavior of the GP tree dramatically. This phenomenon can be traced to the weak causality of GP (Rosca, 1995), which means that small alterations in the underlying structure of a GP tree cause big changes in the behavior of the GP tree. So measuring a sort of "Hamming" distance between the structures of two GP trees to predict the difference of the behavior/function is not well founded, and thus inefficient. This makes a useful definition of a sharing radius hard to determine. It seems that distance metrics in the structure space and the parameter space and the association of a set of parameters with the structure to which they apply must be faithfully captured in order to most effectively maintain both structure diversity and parameter diversity and thereby to achieve efficient search. Therefore, given the inherent difficulty of structure/function mapping, perhaps it is counterproductive to use any structural similarity measure beyond the most basic and completely faithful one – the identity mapping: two structures are either identical, or they are not. That is the structural distance measure used here. While it is possible to define a broader relationship that still captures identity of function (for example, if swapping of the order of two children of a node has no effect on the function computed), such definitions depend on the semantics of the functions, and were not implemented here.

3 BALANCED STRUCTURE AND PARAMETER SEARCH IN EVOLUTIONARY DESIGN BY GP

In design problems involving both variable structure and variable parameters, search must be balanced between the structure and parameters. On one hand, each structure needs sufficient exploration of its parameters to develop its potential to some extent, which means that a reasonable number of individuals of the same structure must probably be kept in the population. On the other hand, no structure should dominate the population, or it would prevent sufficient future exploration of the structure space.

Premature convergence of structures in evolutionary design by GP can be caused by neglecting the different roles of structure and parameter search. In standard GP, nodes at which to perform crossover and mutation are selected randomly from the entire set of nodes, treating those specifying structure modifications identically with those specifying numerical modifications (provided that numerical subtrees are used to define the parameters of components, as is often done). This means that a new circuit structure is often discarded by the selection process if its fitness is low with its initial set of parameters. The result is that often, structures of moderate quality with slightly better parameters proliferate and dominate the population, while inherently better structures with bad parameters are discarded. This is called the *premature structural convergence* problem. This phenomenon arises from the fact that ''promising'' structures are often discarded just because their current parameters are not adjusted well enough to demonstrate their potential. Ideally, a structure should be discarded only when it is demonstrated to be bad after a sufficient effort to adjust its

parameters. In addition, since there are often many more numeric nodes than structure modifying nodes, premature structural convergence is accentuated, since there is a lower probability of choosing a structure modifying node that will generate a new structure than of choosing a node that changes only parameters.

In order to address this problem, structure and parameter search must be controlled explicitly. In our work, a probabilistic method is first devised to decide whether GP does a structure modification (crossover or mutation on a structure modifying node) or does parameter modification (crossover or mutation on a parameter modifying node). Since structure changes have a more fundamental effect than the parameter changes on the performance of the system, we introduce a bias toward more parameter modifications than structure modifications by controlling the probability of selecting these types of nodes for crossover and mutation sites. The following example probabilities are defined to facilitate keeping the structure and its function stable and to allow parameters to be adjusted well enough to demonstrate the potential of a structure.

$$p(\textit{structure modification}) = 0.1$$
$$p(\textit{parameter modification}) = 0.9$$

We also use explicit control of the node selection process to achieve balanced parameter evolution for all parameters in a structure. During the parameter modification stage, we first establish a list of all variables whose values need to be established during evolution, then we randomly select a variable as the current variable to be changed. We then select a node in the numeric subtree of this variable and do a crossover or mutation operation. In this way, each variable has an equal opportunity to be changed during evolution. This improvement speeds the evolution of balanced numeric subtrees. All variables tend to have numeric subtrees with similar depths.

Even with the methods above, premature structural convergence still often occurs as structures with well-fitted parameters quickly dominate the whole population. The Structure Fitness Sharing (SFS) method is proposed to control the reproduction of high-fitness structures. Our assumptions are that fitness sharing can profitably be based on the number of individuals with the same structure, and that distance between the structures of two GP trees with distinct structures is not generally an adequate predictor of the differences between their behaviors. Thus, any "counting of positions where the trees differ" distance metric is not well founded. Instead, a simple labeling technique is used to distinguish structures.

4 STRUCTURE FITNESSS SHARING (SFS)

Structure Fitness Sharing is the application of fitness sharing to structures in GP. In contrast to the GA fitness sharing using a distance measure to identify peaks, in SFS, fitness sharing is based on the tree structures, treating each tree structure in GP as a peak in the space of parameters and structures.

In SFS, each structure is uniquely labeled, whenever it is first created. So long as GP operations on an individual do not change its structure, but only its parameters, the structure label of this individual is not changed. Parameter crossover and mutation, or replication of the individual, simply increase the number of individuals with this structure (label) in the population. If structure modifications are conducted on an individual that change the structure – for example, we change a Rep_C (a GP function node replacing a resistor or inductor of the circuit with a capacitor) to a Rep_I (a GP function replacing a resistor or capacitor of the circuit with an inductor) node – then a new structure label (structureID) is created and is attached to this new individual. Our assumption is that the possibility that any particular structure-altering operation produces exactly the same structure possessed by other individuals in the current population is relatively low, so it is not necessary (or worthwhile) to check a new structure against all other existing structures to see if it is identical with one of them (and so could use its label), although a hashing technique might make this relatively easy to do. Furthermore, even if some newly created individual shares the same structure with another individual but is labeled with a different structure label, the algorithm is not strongly affected, so long as it occurs infrequently.

In standard GP, individuals with certain structures will prosper while others will disappear because of their low fitnesses. If this process is allowed to continue without control, some good structures (usually one) tend to dominate the population and premature convergence occurs. To maintain diversity of structures, fitness sharing is applied to individuals of each structure. SFS decreases the fitness of the individual as follows: SFS penalizes only those structures having too many individuals, according to the following fitness adjustment rule used for the experiments in this paper:

N_s : Number of structures to be searched simultaneously

N_{esp} : Expected number of search points (individuals) for each structure in the whole population

$N^{s_i}_{Ind_i \in s_i}$: Number of individuals with structure s_i (of which individual ind_i is one)

For each individual Ind_i if $N^{s_i}_{Ind_i \in s_i} > 0.8 * N_{esp}$ then

$$k = \left(\frac{N^{S_i}}{\frac{Ind_i \in S_i}{N_{esp}}} \right)^{-\alpha} \quad \text{where } \alpha = 1.5$$

If $k > 1$ *then* $k = 1.0$

$$f_{adj} = f_{old} \quad * \quad k \qquad (1)$$

With this method, each structure has a chance to do parameter search. Premature convergence of structures is limited, and we can still devote more effort to high-fitness structure search.

4.1 LABELING TECHNIQUE IN SFS

A labeling technique is used in SFS to distinguish different structures efficiently. A similar technique has been used by Spears (1994), in which tag bits are used to identify different subpopulations. Spears's result suggests that in crowding and fitness sharing in GA, we only need to decide whether or not two individuals have the same label. The added precision of the distance metric for maintaining the diverse state of a subpopulation is often unnecessary. In SFS, the label is used only to decide whether or not two individuals have the same label (*i.e.,* structure). We use simple integer numbers as labels rather than more complicated tag bits. Ryan (1994, 1995) also uses similar labelling ideas to decide which race an individual belongs to in his racial GA (RGA).

4.2 HASH TABLE TECHNIQUE IN SFS

In order to keep track of all individuals with each particular label, SFS uses a hash table is used -- this speeds up the access to the structure by each individual when we do crossover, mutation, and reproduction. Each time we create a new structure, we create an entry in the hashtable with the structureID (next integer) as the key. The size of the hash table is controlled to accommodate at most 500 structures in our experiments. Whenever the number of structure entries in the hashtable exceeds 500, those structure entries with no individuals in the current population or with a low fitness of its best individuals and with old ages (generations since their labels were created) are removed from the hashtable. The corresponding individuals of these structures are removed from the current population at the same time.

4.3 THE STRUCTURE FITNESS SHARING ALGORITHM IN GP

The following is the outline of the algorithm of SFS as applied to GP:

Step 1: Initialize the population with randomly generated individuals. Initialize the structure hash table.

Step 2: Assign each individual a unique label. Here a label is just an unassigned integer number incremented by one at each assignment.

Step 3: Loop over generations

3.1 Select the parents for a genetic operation according to their standard fitness

3.2: If current operation is an operation that changes the structure from that of the parent(s), (including crossover and mutation at structure operator nodes of GP trees)

Create a new label for each new structure created and add the new structure item to the structure hash table.

3.3: If the current operation is a parameter modification (mutating the parameter nodes or crossing over at a parameter node) or only replication of an existing individual, do not create a new label. New individuals inherit the labels from their parents. Update information about the structure items in the hash table, including the best fitness of this structure, number of individuals, age, etc.

3.4: If the maximum number of structures in the hash table is reached, first purge those structures that have no individuals in the current population. If there are still too many structures, then delete those structures whose best individuals have lower fitness and high age (>10 generation, in our case) and delete their individuals in the population and replace them with new individuals formed by crossover or mutation until the maximum number of structures in hash table is kept.

3.5: Adjust the fitness of each individual according to equation (1).

Step 4: If stopping criterion is satisfied, stop; else go to step 3.

5 EXPERIMENTS

5.1 PROBLEM DEFINITION

GP with the SFS technique has been applied to an analog circuit synthesis problem that was previously approached using GP without SFS (Rosenberg, 2001). In this problem, an analog circuit is represented by a bond graph model (Fan, 2001) and is composed of inductors (I), resistors (R), capacitors (C), transformers (TF), gyrators (GY), and sources of effort (SE). The developmental evolution method similar to (Koza, 1999) is used to evolve both the structure and parameters of a bond graph representation of a circuit that achieves the user-specified behavior. With this method, a set of structure modifying operators (e.g. Rep_C, Rep_I) are provided to operate on the embryo

bond graph. A set of numeric operators (such as +, -, *, /) are provided to modify the parameters of the structure.

In this paper, the design objective is to evolve an analog circuit with response properties characterized by a pre-specified set of eigenvalues of the system equation. By increasing the number of eigenvalues specified, we can define a series of synthesis problems of increasing difficulty, in which premature convergence problems become more and more significant when traditional GP methods are used. This eigenvalue assignment problem has received a great deal of attention in control system design. Control over eigenvalues in designing systems, in order to avoid instability and to provide particular response characteristics, is often an important and practical problem.

Circuit synthesis by GP is a well-studied problem that generally demands large computational power to achieve good results. Since both the topology and the parameters of a circuit affect its performance, it is easy to get stuck in the evolution process.

5.2 EXPERIMENTAL SETUP

In the example that follows, a set of target eigenvalues was given and a bond graph model with those eigenvalues was generated. Table 1 shows the three sets of 6, 8, and 10 target eigenvalues used as targets for example genetic programming runs:

We applied single population GP with and without SFS and multi-population GP with and without SFS to all three problem instances. Each experiment was repeated 10 times with different random seeds.

Table 1. Target Eigenvalues

Problem 1:	6-eigenvalue problem
$-0.1 \pm 5.0j, -1.0 \pm 2.0j, -2.0 \pm 1.0j$	
Problem 2:	8-eigenvalue problem
$-0.1 \pm 5.0j, -1.0 \pm 2.0j, -2.0 \pm 1.0j, -3.0 \pm 0.7j$	
Problem 3:	10-eigenvalue problem
$-0.1 \pm 5.0j, -1.0 \pm 2.0j, -2.0 \pm 1.0j, -3.0 \pm 0.7j$ $-4.0 \pm 0.4j$	

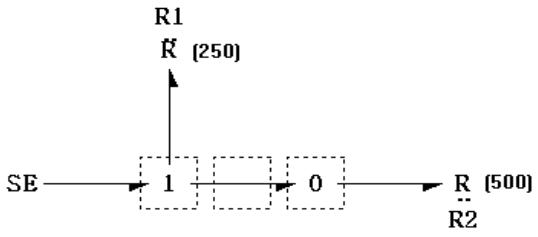


Figure 1. The Embryo Bond Graph Model

The embryo model used is shown in Figure 1. It represents an embryo bond graph with three initial modifiable sites (represented as dotted boxes). In each case, the fixed components of the embryo are sufficient to allow definition of the system input and output, yielding a system for which the eigenvalues can be evaluated, including appropriate impedances. The construction steps specified in the GP tree are executed beginning from this embryo. The numbers in parentheses represent the parameter values of the elements.

Three circuits of increasing complexity are to be synthesized, with eigenvalue sets as specified above. The GP parameter tableau for the single population method is shown in Table 2 below.

These problems exhibit a very high degree of epistasis, as a change in the placement of any pair of eigenvalues has a

Table 2. Parameter Settings for GP

Parameters of Single Population GP	Popsize: 1000 init.method = half_and_half init.depth = 3-6 max_nodes = 1000 max_depth = 15 crossover rate = 0.9 mutation rate = 0.1 max_generation = 1000
Additional Parameters of Multi-Population GP	Number of subpopulations = 10; Size of subpop = 100 migration interval = 10 generations migration strategy: ring topology, migrate 10 best individuals to the next subpopulation in the ring to replace its 10 worst individuals
SFS Parameters	N_s : 50 N_{esp} : 20 = popsize/ N_s

strong effect on the location of the remaining eigenvalues. Eigenvalue placement is very different from "one-max" or additively decomposable optimization problems, and these problems become increasingly difficult with the problem order. The performance of each of the three GP approaches is reported in Figure 2, in which the four GP methods are indicated by

OneGP: single population GP, no SFS

MulGP: multi-population GP, no SFS

ONE.SFS: single population GP with SFS

MULPOP.SFS: multi-population GP with SFS

To observe the effect of structure fitness sharing, we monitor the number of distinct structures in the experiments with and without SFS techniques. From Fig 2, one can see that Structure Fitness Sharing can significantly improve the performance for single population GP and also does better in multi-population GP,

though the difference is not as significant. The reason may be that multi-population runs already provide an inherent diversity maintenance mechanism. We also find that SFS can help to provide probabilistic control of structure and parameter modifications to maintain a stable number of search structures in the whole population, as illustrated in Fig 3.

6 CONCLUSIONS

In this paper, Structure Fitness Sharing (SFS) is proposed to achieve balanced structure and parameter search in evolutionary design by Genetic Programming. SFS can effectively prevent the dominance of any specific structure and when combined with probabilistic control of structure and parameter modification, SFS can maintain a stable number of structures for simultaneous structure and parameter search. The labeling technique in SFS eliminates the necessity of computing the distance between two individuals, which saves computing effort that we believe is often largely wasted when attempting to measure GP structural similarity. The user parameters of the standard fitness sharing method are also eliminated (e.g. the sharing radius). All that must be done is to define the fitness adjustment scheme: how to penalize the fitness of a structure when the number of individuals with that structure label grows large enough to threaten the diversity of the population. The hash table technique allows SFS to quickly update the structure information about the current population during evolution. More complicated balanced structure parameter search methods can be derived using the concept of structure diversity. For example, the authors intend to incorporate the age concept and the elitism method of multi-objective evolutionary computation into SFS. We also intend to explore use of a separate GA for explicit exploration of the parameter spaces of individual structures, with the expectation of a significant impact on the selection, crossover and mutation dynamics of the overarching simultaneous evolutionary search of structure and parameters.

Acknowledgment

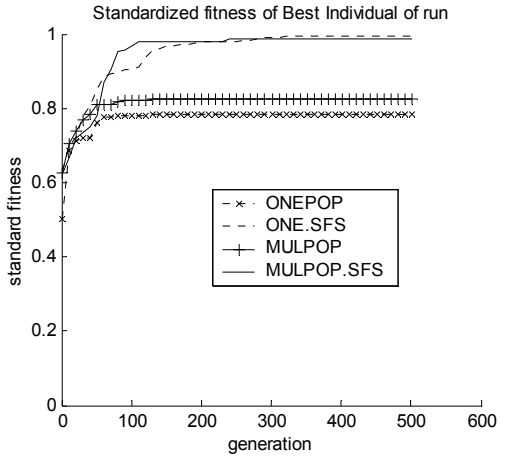
Thanks to Ahmad R Shahid for his assistance with the hash table technique. This work was supported by the National Science Foundation under contract DMI 0084934.

References:

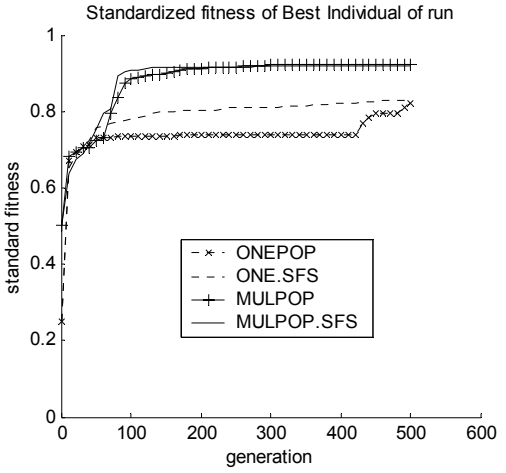
P. Nordin, *Evolutionary Program Induction of Binary Machine Code and Its Application*, PhD thesis, University of Dortmund,Germany, 1997.

L. Spector, H. Barnum, and H. J. Bernstein, “Quantum Computing Applications of Genetic Programming”, *Advances in Genetic Programming 3*, L. Spector *et al*., eds., MIT Press, Cambridge, Mass., 1999, pp.135-160.

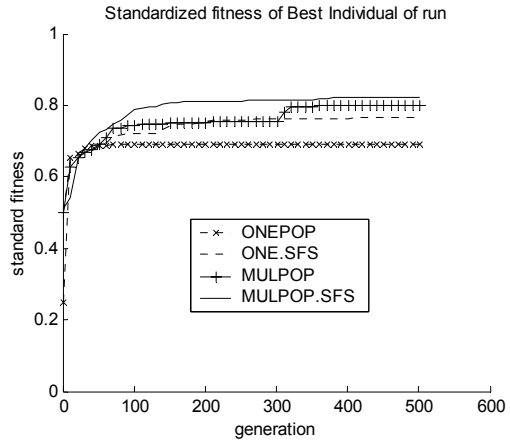
D. Andre and A. Teller, “Evolving Team Darwin United,"



a) 6-eigenvalue problem



b) 8-eigenvalue problem



c) 10-eigenvalue problem

Figure 2. Fitness of Best Individual to Date vs. Generation

RoboCup-98: Robot Soccer World Cup II. Lecture Notes in Computer Science, M. Asada and H. Kitano, eds., Vol. 164, Springer-Verlag, Berlin, pp.346-352, 1999.

J. R. Koza *et al*., *Genetic Programming III: Darwinian Invention and Problem Solving*, Morgan Kaufmann, San Francisco, 1999.

K. Rodriguez-Vazquez, C. M. Fonseca, and P. J. Fleming, "Multiobjective Genetic Programming : A Nonlinear System Identification Application," *Proc. Genetic Programming '97 Conference*, pp. 207-212, 1997

C. M. Fonseca, and P. J. Fleming, "Genetic Algoritms for Multiobjective Optimisation: Formulation, Discussion, and Generalization," *Proceedings Fifth International Conference on Genetic Algorithms*, Morgan Kaufmann, San Francisco, pp. 416-423, 1993.

S. Oliker, M. Furst, and O. Maimon, "A distributed genetic algorithm for neural network design and training," *Complex Systems*, 6, pp. 459-477, 1992.

K. A. DeJong. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*, PhD thesis. University of Michigan. Dissertation Abstracts International 36(10), 5410B. (University Microfilms No. 76--9381), 1975.

E. D. DeJong, R. A. Watson, and J. B. Pollack, "Reducing Bloat and Promoting Diversity using Multi-Objective Methods," *Proc. GECCO-2001*, Morgan Kaufmann, San Francisco, pp. 11-18.

S. W. Mahfoud, "Crowding and Preselection Revisited," *Proc. PPSN-92*, Elsevier, 1992.

D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning,* Addison Wesley, 1989.

D. Beasley, D.R. Bull, and R.R. Martin, "A Sequential Niche Technique for Multimodal Function Optimisation," *Evolutionary Computation*, 1, pp. 101-125. 1993.

A. Ekárt; S. Z. Németh, "A Metric for Genetic Programs and Fitness Sharing," in R. Poli, W. Banzhaf, W. B. Langdon, J. Miller, P. Nordin, T. Fogarty (eds.), *Genetic Programming, Proceedings of EUROGP'2000*, Edinburgh, 15-16 April 2000, LNCS volume 1802, pp. 259-270.

R. I. McKay, "Fitness Sharing in Genetic Programming," *Proc. GECCO-2000*, Las Vegas, NV, Morgan Kaufmann, San Francisco, July, 2000.

J. P. Rosca and D. H. Ballard, "Causality in Genetic Programming," in L. J. Eshelman, ed., *Proc. Sixth International Conference on Genetic Algorithms*, Morgan Kaufmann, San Francisco, pp. 256-263, 1995.

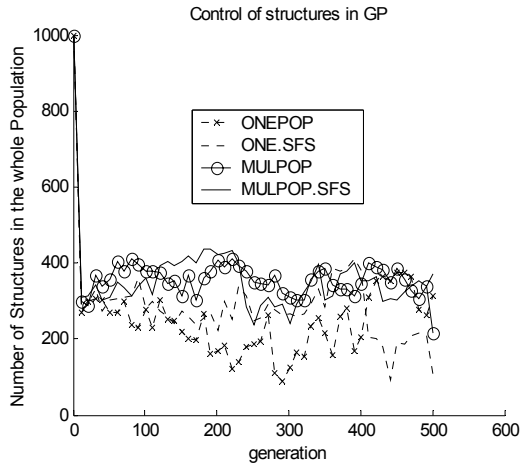
W. M. Spears, "Simple Subpopulation Schemes," *Proc. Evolutionary Programming Conf.*, pp. 296-307, 1994.

C. Ryan, "Pygmies and Civil Servants, in K. E. Kinnear, Jr., ed., *Advances in Genetic Programming*, pp. 243-263. MIT Press, 1994.

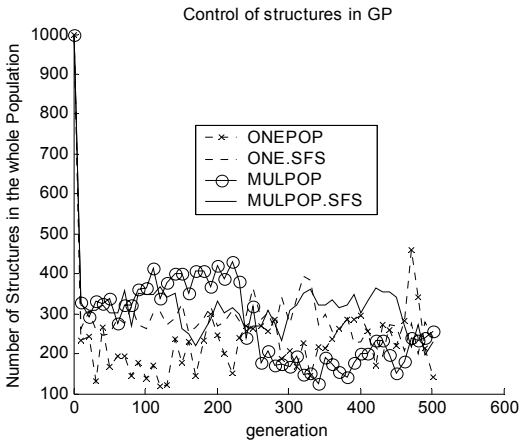
C. Ryan, "Racial Harmony and Function Optimization in Genetic Algorithms - the Races Genetic Algorithm," in *Proc. Of Evolutionary Programming 1995*. pp. 296-307. MIT press.

R.C. Rosenberg, E. D. Goodman and K. Seo, "Some Key Issues in Using Bond Graphs and Genetic Programming for Mechatronic System Design", *Proc. ASME International Mechanical Engineering Congress and Exposition*, 2001, November 11-16, New York.

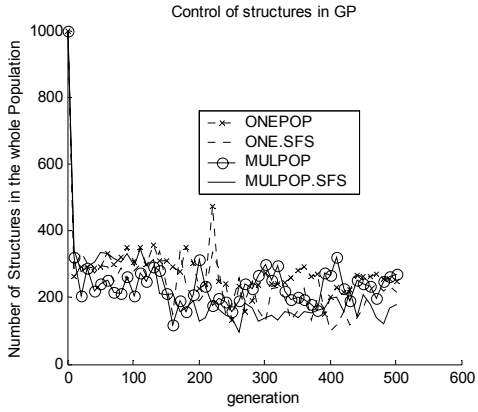
Z. Fan, J. Hu, K. Seo, E. Goodman, R. Rosenberg, and B. Zhang, "Bond Graph Representation and GP for Automated Analog Filter Design," *Genetic and Evolutionary Computation Conference 2001 Late Breaking Papers*, ISGEC Press, San Francisco, pp. 81-86.



a) 6 eigenvalue problem



b) 8 eigenvalue problem



c) 10 eigenvalue problem

Figure 3. Number of Structures in Population vs. Generation

Inference of Differential Equation Models by Genetic Programming

Hitoshi Iba
Grad. School of Frontier Informatics,
The University of Tokyo
7-3-1 Hongo, Bunkyo-ku, 113-8656, Japan

Erina Sakamoto
Grad. School of Inf. and Comm. Eng.,
The University of Tokyo
7-3-1 Hongo, Bunkyo-ku, 113-8656, Japan

Abstract

This paper describes an evolutionary method for identifying a causal model from the observed time series data. We use a system of ordinary differential equations (ODEs) as the causal model. This approach is well-known to be useful for the practical application, e.g., bioinformatics, chemical reaction models, controlling theory etc. To explore the search space more effectively in the course of evolution, the right-hand sides of ODEs are inferred by Genetic Programming (GP) and the least mean square (LMS) method is used along with the ordinary GP. We apply our method to several target tasks and empirically show how successfully GP infers the systems of ODEs.

1 Introduction

Ordinary differential equations (ODEs) are one of the easiest media for modeling complex systems, where basic differential relationships are known between the system components. Solving a set of differential equations to produce their equivalent functions is relatively easy so as to obtain useful time-series data. On the other hand, the inverse problem, i.e., the inference of the system of ODE from the observed time-series data, is not necessarily easy, although very important for many fields. This is because there is no knowing the appropriate form, i.e., the order and terms of ODEs, beforehand.

In this paper, we deal with an arbitrary form in the right-hand side of the system of ODEs to allow the flexibility of the model. More precisely, we consider the following general form:

$$\frac{dX_i}{dt} = f_i(X_1, X_2, \ldots, X_n) \quad (i = 1, 2, \ldots, n), \qquad (1)$$

where X_i is the state variable and n is the number of the observable components.

For the sake of identifying the system, we use Genetic Programming (GP) to evolve the ODEs from the observed time series. Although GP is effective in finding the suitable structure, it is sometimes difficult to optimize the parameters, such as constants or coefficients of the polynomials. This is because the ordinary GP searches for them simply by combining randomly generated constants. To avoid this difficulty, we introduce the least mean square (LMS) method.

There have been several studies for identifying differential equation models by means of EAs (Evolutionary Algorithms). For instance, GP was used to find a function in a symbolic form, which satisfies the differential equation and initial conditions [Koza92]. Cao and his colleagues used hybrid evolutionary modeling algorithms [Cao00]. The main idea was to embed GA in GP, where GP was employed to discover and optimize the structure of a model, while GA was used to optimize its parameters, i.e., coefficients. [Babovic00] also applied GP to approximate several ODEs from the domain of ecological modeling, e.g., Lotka-Volterra and logistic equations. They showed that the GP-based approach introduced numerous advantages over the most available modeling methods. In our previous researches [Sakamoto and Iba00] and [Sakamoto and Iba01], we proposed another integrated scheme, in which the least mean square (LMS) method is used along with GP. In this scheme, some individuals were created by the LMS method at some intervals of generations and they replaced the worst individuals in the population.

In this paper, we extend our previous approach so

as to achieve the inference of the ODEs more effectively. More precisely, we empirically show the following points:

- The success in the acquisition of ODEs, which are close to the observed time series.
- The inference of the exact equation form, i.e., the exact causal relationship.
- The effectiveness of the LMS method.
- The superiority of our approach over the previous methods.

The rest of this paper is organized as follows. In Section 2, we describe the details of our method, i.e., how GP and LMS methods are integrated to work in the course of evolution. Three examples are used to examine the effectiveness of our method. Their experimental results are shown in Section 3. Then, we discuss the results in Section 4 and give some conclusion in Section 5.

2 Integration of GP and LMS

We use GP to identify a causal model in the form of the system of ODEs. Though GP is capable of finding a desirable structure effectively, it cannot always be effective in finding the proper coefficients because GP uses the combination of randomly selected ones. We have chosen the least mean square method (LMS) to tackle this defect of the ordinary GP. For this purpose, coefficients are not included in the terminal set for a GP individual tree. The coefficients of each term of a GP tree are calculated by the LMS method and a table of them composes a GP individual along with a tree.

2.1 Inference of the form of equations using GP

We use GP to identify the form of the system of differential equations. For this purpose, we encode right-hand sides of ODEs into a GP individual. Each individual contains a set of n trees, i.e., an n-tuple of trees$(f_1, \ldots, f_n)$. For example, consider the two trees in Fig.1. This shows the following system of ODEs:

$$\begin{cases} \dot{X}_1 = & aX_1X_2^2 + b \\ \dot{X}_2 = & cX_1X_2 + dX_2, \end{cases} \tag{2}$$

where the coefficients a, b, c, d, are derived by LMS described later. Note that the constant term b is added to the right hand side of the first equation, because of the constant terminal, i.e., 1. Thus, each equation uses a distinct program. A GP individual maintains multiple branches, each of which serves as the right-hand side of a differential equation.

Crossover operations are restricted to the correspondent branch pairs. Actually, each tree, i.e., each right hand side of the ODE sytem, is evolved independently in parallel.

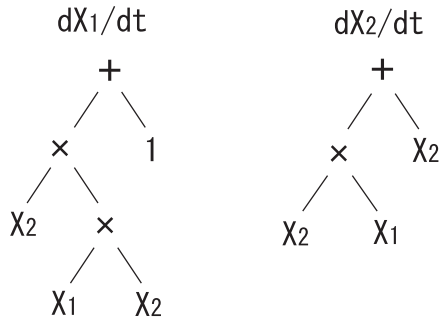


Figure 1: Example of a GP individual.

2.2 Optimization of models using LMS method

Coefficients of a GP individual is derived by the LMS method described below. Assume that we want to acquire the approximate expression in the following form:

$$y(x_1, \ldots, x_L) = \sum_{k=1}^{M} a_k F_k(x_1, \ldots, x_L), \tag{3}$$

where $F_k(x_1, \ldots, x_L)$ is the basis function, $x_1, \ldots, x_L$ are the independent variables, $y(x_1, \ldots, x_L)$ is the dependent variable, and M is the number of the basis functions. Let $\mathbf{a}$ be the vector of coefficients, i.e., $(a_1, \ldots, a_M)$. Then, our purpose is to minimize χ^2 described in (4) to acquire $\mathbf{a}$.

$$\chi^2 = \sum_{i=1}^{N} \Big(y(i) - \sum_{k=1}^{M} a_k F_k(x_1(i), \ldots, x_L(i)) \Big)^2, \tag{4}$$

where $x_1(i), \ldots, x_L(i)$ and $y(i)$ are data given for the LMS method and N is the number of data points. Let $\mathbf{b}$ be the vector of $(y(1), \ldots, y(N))$ and $\mathbf{A}$ be the $N \times M$ matrix described below:

$$\begin{pmatrix} F_1(x_1(1), \ldots, x_L(1)) & \ldots & F_M(x_1(1), \ldots, x_L(1)) \\ F_1(x_1(2), \ldots, x_L(2)) & \ldots & F_M(x_1(2), \ldots, x_L(2)) \\ \vdots & \ddots & \vdots \\ F_1(x_1(N), \ldots, x_L(N)) & \ldots & F_M(x_1(N), \ldots, x_L(N)) \end{pmatrix}$$

Then, (5) should be satisfied to minimize χ^2.

$$(\mathbf{A^T} \cdot \mathbf{A}) \cdot \mathbf{a} = \mathbf{A^T} \cdot \mathbf{b} \tag{5}$$

Thus, $\mathbf{a}$ can be acquired by solving this equation.

When applying to the time-series problem, $y(i)$ for the jth equation of the system of differential equations is calculated according to the following discrete difference of the time-series $x_j(t)$:

$$y(i) = \dot{X}_j|_{t=t_i} = \frac{x_j(t_i + \Delta t) - x_j(t_i - \Delta t)}{2\Delta t}, \tag{6}$$

where t_i is the time of the ith selected data point. For example, consider the first ODE ($\dot{X}_1$) of the system (2), in which the number of the components is two ($L = n = 2$). In this case, we are using two basis functions, i.e., $M = 2$ and (F_1, F_2)=$(X_1 X_2^2, 1)$. Then, the ith row of the matrix $\mathbf{A}$ is determined as $(x_1(t_i)x_2(t_i)^2$,1).

The coefficients in the approximate expressions of the right-hand sides of the equations can be derived by using $\mathbf{A}$ and $\mathbf{b}(y(1), \ldots, y(N))$ acquired above.

2.3 Fitness definition

The fitness of each individual is defined as the sum of the squared error and the penalty for the degree of the equations:

$$fitness = \sum_{i=1}^{n} \sum_{k=0}^{T-1} (x'_i(t_0 + k\Delta t) - x_i(t_0 + k\Delta t))^2 + a \cdot m, \tag{7}$$

$$\left(\begin{array}{lcl} t_0 & : & \text{the starting time} \\ \Delta t & : & \text{the stepsize} \\ n & : & \text{the number of the observable components} \\ T & : & \text{the number of the data points} \end{array} \right)$$

where $x_i(t_0 + k\Delta t)$ is the given target time series ($k = 0, 1, \cdots, T-1$). $x'_i(t_0 + k\Delta t)$ is the time series acquired by calculating the system of ODEs represented by a GP individual. All these time series are calculated by using the forth-order Runge-Kutta method. m is the number of terms and a is the weight constant. In other words, the individual which has a smaller number of terms and is closer to the target time series has the higher possibility to be selected and inherited to the next generation. This fitness derivation is based on the MDL (Minimum Description Length) criterion, which has been often used in GP (see [Iba94], [Zhang95] and [Nikolaev and Iba01] for examples). When calculating the time series, some individuals may go overflow. In this case, the individual's fitness value gets so large that it will be weeded out from the population.

	Exp.1	Exp.2	Exp.3	Exp.4
Population size	1000	1000	1000	3000
Generation	100	100	100	100
Crossover rate	0.80	0.80	0.80	0.80
Mutation rate	0.10	0.10	0.10	0.10
# time series	1	1	3	3
Stepsize	0.01		0.01	0.01
# data points	100		40	30

Table 1: GP and LMS parameters for experiments.

We use several sets of time series as the training data for GP. This is to acquire the equations as close to the target as possible. Each data set was generated from the same target by using different initial values.

3 Experimental results

We have prepared three different tasks to test the effectiveness of our method. Experimental parameters are summarized in Table 1. Function and terminal sets F and T are as follows:

$$\begin{array}{ll} F = & \{+, -, *\} \\ T = & \{X_1, \ldots, X_n, 1\} \end{array}$$

3.1 Example 1 : Chemical reaction model

The reaction between formaldehyde (X_1) and carbamide in the aqueous solution gives methylol urea (X_2) which continues to react with carbamide and form methylene urea (X_3) (see [Cao00] for details). The reaction equations are described as below:

$$HCHO + (NH_2)_2CO \xrightarrow{k_1} H_2N \cdot CO \cdot NH \cdot CH_2OH \tag{8}$$

$$H_2N \cdot CO \cdot NH \cdot CH_2OH + (NH_2)_2CO \xrightarrow{k_1} (NH_2CONH)_2CH_2 \tag{9}$$

As a kind of typical consecutive reaction, the concentrations of the three components in the system satisfy the following system:

$$\begin{cases} \dot{X}_1 = & -1.4000X_1 \\ \dot{X}_2 = & 1.4000X_1 - 4.2000X_2 \\ \dot{X}_3 = & 4.2000X_2 \end{cases} \tag{10}$$

Experimental parameters for this task are shown in Table 1. By applying our method, we have acquired the system of eq.(11), which gave the sums of square errors as $(X_1, X_2, X_3) = (0.000, 2.082 * 10^{-11}, 1.883 * 10^{-11})$. The time series generated by this system is shown in Fig.2 along with that of the target.

$$\begin{cases} \dot{X}_1 = & -1.4000X_1 \\ \dot{X}_2 = & 1.4004X_1 - 4.2006X_2 \\ \dot{X}_3 = & 4.1998X_2 \end{cases} \quad (11)$$

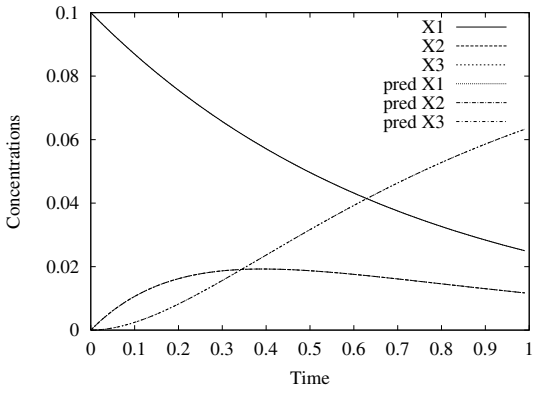


Figure 2: Time series of the acquired model for chemical reaction.

The best kinetic model acquired in [Cao00] was as follows:

$$\begin{cases} \dot{X}_1 = & -1.400035X_1 \\ \dot{X}_2 = & 1.355543(X_1 + t) - 4.482911X_2 \\ \dot{X}_3 = & 4.069420X_2 + t - 0.002812 \end{cases} \quad (12)$$

where the sums of square errors were $(X_1, X_2, X_3) = (1.600*10^{-11}, 3.240*10^{-8}, 3.025*10^{-9})$. Note that the terminal set used in [Cao00] included the time variable t.

3.2 Example 2 : Three-species Lotka-Volterra model

The Lotka-Volterra model describes interactions between two or more species, i.e., predators and preys, in an ecosystem [Takeuchi96]. The following DOEs represent a three-species Lotka-Volterra model:

$$\begin{cases} \dot{X}_1 = & (1 - X_1 - X_2 - 10X_3)X_1 \\ \dot{X}_2 = & (0.992 - 1.5X_1 - X_2 - X_3)X_2 \\ \dot{X}_3 = & (-1.2 + 5X_1 + 0.5X_2)X_3 \end{cases} \quad (13)$$

This system models the introduction of the third species, i.e., a predator, into a two-species system of competition, i.e., preys. More precisely, X_1 and X_2 are the number of preys competing with each other, whereas X_3 represents the number of predators.

The GP and LMS parameters we used are shown in Table 1. As a result of experiments, the following DOEs were acquired in a typical run:

$$\begin{cases} \dot{X}_1 = & -10.001X_1X_3 - 1.000X_1X_2 - 0.999X_1^2 + 1.000X_1 \\ \dot{X}_2 = & 0.992X_2 - 1.500X_1X_2 - 0.996X_2X_3 - 1.000X_2^2 \\ \dot{X}_3 = & 4.998X_1X_3 + 0.500X_2X_3 - 1.200X_3 \end{cases} \quad (14)$$

Note that the two systems of DOEs, i.e., eqs.(13) and eqs.(14), are almost identical except for slightly different coefficients. In all runs, we have succeeded in acquiring almost the same DOEs. The MES (Mean Square Error) of the above DOEs are very small $(4.78 * 10^{-11})$.

We have conducted the further experiments with this Lotka-Volterra model to compare the performances of the following methods:

- Standard GP
- Old version of GP with LMS
- Proposed method of GP with LMS

As mentioned in Section 1, in our previous papers [Sakamoto and Iba01] and [Sakamoto and Iba00], we used the least mean square (LMS) method along with GP in a different way, i.e., some individuals were created by the LMS method at some intervals of generations and they replaced the worst individuals in the population. We compared the performance of the old version to see the effectiveness of the approach proposed in this paper.

The experimental results are given in Table 2. The table shows the MSE data and hit percentages, i.e., the ratios of successes in acquiring the target DOEs, averaged over ten runs. As clearly shown in the table, GP with LMS performed better than GP alone (standard GP), in view of MSE values. Moreover, the superiority of the proposed approach over the old version has been confirmed by the hit percentage.

3.3 Example 3 : E-cell simulation

We have conducted the experiment on the data of a metabolic network that consists of three substances.

	MSE	Hit(%)
Standard GP	$4.47 * 10^{-5}$	0%
Old version	$2.85 * 10^{-7}$	0%
Proposed method	$4.78 * 10^{-11}$	100%

Table 2: Comparision of Three methods.

This target network is a part of the biological phospholipid pathway. The data were derived from the E-cell simulation model. E-cell Simulation Environment (E-CELL SE) is a software package for cellular and biochemical modeling and simulation (see [Tomita99] for details of bioinformatics). This network can be approximated as (15).

$$\begin{cases} \dot{X_1} = & -k_1 X_1 X_3 \\ \dot{X_2} = & k_1 X_1 X_3 - k_2 X_2 \\ \dot{X_3} = & -k_1 X_1 X_3 + k_2 X_2 \end{cases} \quad (15)$$

Note that the parameters k_1, k_2, and k_3 are unknown for the simulation experiment.

Three sets of time series generated by E-cell with a different initial value were used for the training of GP. Experimental parameters are shown in Table 1. By applying our method, we have acquired the following equations in a typical run:

$$\begin{cases} \dot{X_1} = & -10.3176 X_1 X_3 \\ \dot{X_2} = & 9.7149 X_1 X_3 - 17.5084 X_2 \\ \dot{X_3} = & -9.7018 X_1 X_3 + 17.4766 X_2 \end{cases} \quad (16)$$

When we compare the two systems, i.e., eq.(16) and eq.(15), we can confirm the success in acquiring the almost identical model to the target ODEs. The time series generated by eq.(16) is shown in Fig.3 along with that of the target. The average MSE (Mean Square Error) of 10 runs was $2.545 * 10^{-3}$.

We have also conducted a comparative experiment without the LMS method to confirm its effectiveness (in this case, coefficients are added to the terminal set). The average MSE of 10 runs is $5.328 * 10^{-3}$, whereas that of the experiment with the LMS method is $2.545 * 10^{-3}$. Besides, the correct form of ODEs was not always acquired without the LMS method. For example, in no runs, the correct ODE for X3 was acquired without the LMS method.

3.4 Example 4 : S-system model

S-system is a type of power-law formalism and has been proposed for the causality model. The concrete

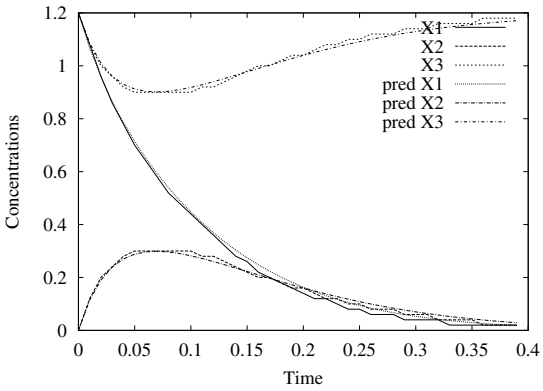


Figure 3: Time series of the acquired model for E-cell simulation.

form of S-system is given as follows:

$$\frac{dX_i}{dt} = \alpha_i \prod_{j=1}^{n} X_j^{g_{ij}} - \beta_i \prod_{j=1}^{n} X_j^{h_{ij}} (i = 1, 2, \ldots, n),$$

where X_i is a state variable. The first term represents all influences that increase X_i, whereas the second term represents all the influences that decrease X_i [Savageau76]. S-system is commonly used in many fields and its parameters were optimized by using GA [Tominaga00].

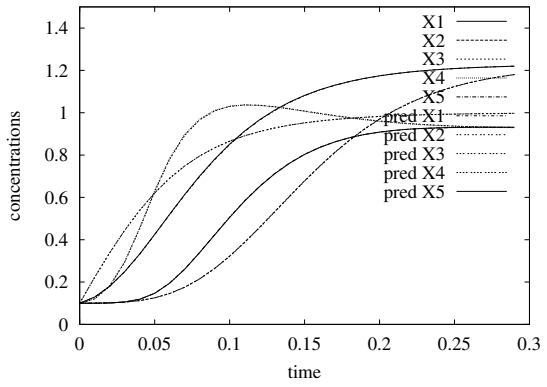
We tested on the gene regulatory network which consists of five nodes and had been generated from the S-system. This causality model can be approximated as follows (see [Tominaga00] for details):

$$\begin{cases} \dot{X_1} = & 15.0 X_3 X_5^{-0.1} - 10.0 X_1^{2.0} \\ \dot{X_2} = & 10.0 X_1^{2.0} - 10.0 X_2^{2.0} \\ \dot{X_3} = & 10.0 X_2^{-0.1} - 10.0 X_2^{-0.1} X_3^{2.0} \\ \dot{X_4} = & 8.0 X_1^{2.0} X_5^{-1.0} - 10.0 X_4^{2.0} \\ \dot{X_5} = & 10.0 X_4^{2.0} - 10.0 X_5^{2.0} \end{cases} \quad (17)$$

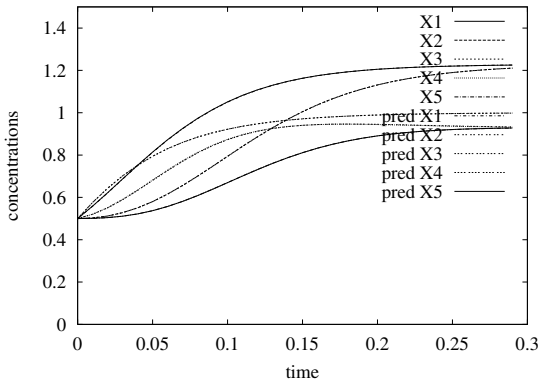
Three sets of time series with a different initial value were used for the training of GP. Experimental parameters are shown in Table 1. To cope with the real-valued power of the component variables, we used the following terminal set:

$$T = \{X_1, X_1^{-1}, X_1^{0.1}, X_1^{-0.1}, X_2, X_2^{-1}, X_2^{0.1}, X_2^{-0.1}, \cdots, X_5, X_5^{-1}, X_5^{0.1}, X_5^{-0.1}\} \quad (18)$$

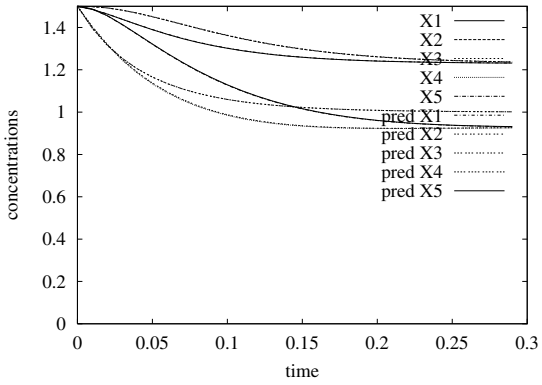
By applying our method, we have acquired the following equations in a typical run:



(a) Data no.1



(b) Data no.2



(c) Data no.3

Figure 4: Acquired and target time series data of S-system.

$$\begin{cases} \dot{X}_1 = & 14.926X_3X_5^{-0.1} - 9.941X_1^{2.0} \\ \dot{X}_2 = & 9.950X_1^{2.0} - 9.938X_2^{2.0} \\ \dot{X}_3 = & 10.010X_2^{-0.1} - 10.005X_2^{-0.1}X_3^{2.0} \\ \dot{X}_4 = & 7.880X_1^{2.0}X_5^{-1.0} - 9.826X_4^{2.0} \\ \dot{X}_5 = & 9.935X_4^{2.0} - 9.919X_5^{2.0} \end{cases} \quad (19)$$

Note that two systems, i.e., eq.(19) and eq.(17), are almost identical. The acquired and the given target time series are shown in Fig.4. As can be seen, the acquired time series is quite close to the target one.

For the above task, the average MSE (Mean Square Error) of 10 runs was $4.532 * 10^{-6}$. On the other hand, that of the experiment without the LMS method was $6.145 * 10^{-4}$. The equations of the correct forms were acquired in 92% of the runs with LMS, whereas in no runs the correct form of equations was acquired without the LMS method. Fig.5 shows the fitness transitions for both methods in typical cases. Thus, we can confirm that the search became more effective by using GP along with LMS method.

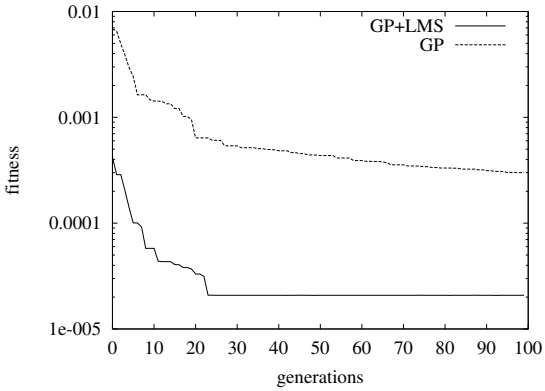
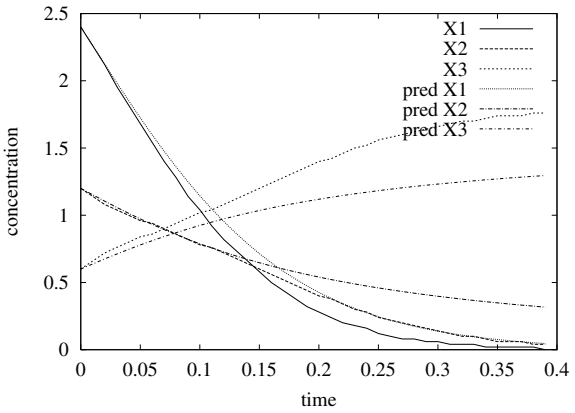


Figure 5: Typical case of the evolution for S-sytem.

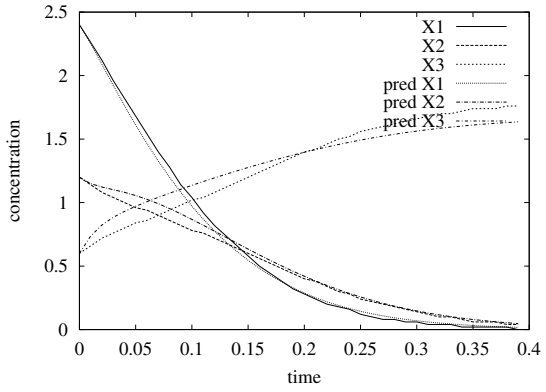
4 Discussion

Although the above section shows the effectiveness of our approach in acquiring the exact form which is very close to the target observed data, there is another

factor to be considered, i.e., the robustness. To test the robustness of our method to the real noisy world, we conducted the E-cell experiment (i.e., Exp.2) with noise-added data sets. 5% and 10% random noises were added to the target time series. The acquired time series are plotted in Fig.6 with the target data. The MSE values and the success ratios averaged over 15 runs are shown in Table 3. The table compares these values by our approach and the standard GP, in which the right hand sides of ODEs are evolved in a similar way to the symbolic regression (see [Koza92] for details). We can observe that the proposed method worked effectively to acquire the better individual with noisy environments than the standard GP.



(a) Standard GP.



(b) Proposed method.

Figure 6: Acquired time series of noisy data.

As can be seen in Section3.1, the proposed approach is superior to the traditional method [Cao00]. Remember that the coefficients of ODEs were derived by means of GA in Cao's scheme, whereas we used LMS for this purpose. Therefore, the superiority of our approach can also be confirmed when we consider the difference of computational burden of these techniques.

As with many other proposed models, the solution which fits the given time series quite well is not necessarily determined uniquely. In other words, there may exist more than one solution which behave consistently with the target. Therefore, even if one system of ODEs is acquired as a solution, we cannot disregard other candidates. Our aim is to obtain the candidates scattered in the huge search space and to propose to users the possible causal relationship among the observable components. Therefore, as future works, we will concentrate on the construction of the interactive system, which proposes the possible solutions and tells users what kinds of data are needed to determine the relationship among the components (see [Mimura and Iba02] for details).

5 Conclusion

We have proposed the inference method of the system of ODEs from the observed time series by using GP along with the LMS method. We showed how successfully our method can infer the causal model by several experiments. More precisely, we succeeded in acquiring the system of ODEs which is very close to the observed time series and inferring the exact equation form. The effectiveness of the LMS method and the superiority of our approach over the previous method were confirmed by comparative experiments.

As a future research, we will apply our approach to some real-world tasks. For this purpose, we are working on the development of the interactive inference system, in which users will be able to pick up the correct equations or discard the meaningless equations from the suggested ones. We are trying to solve some of the real biological problems by using this system [Mimura and Iba02].

References

[Babovic00] Babovic, V., and Keijzer, M., Evolutionary algorithms approach to induction of differential equations, in Proc. of *Genetic and Evolutionary Computation Conference (GECCO2000)*, pp.251-258, 2000.

[Cao00] Cao, H., Kang, L., Chen, Y., Yu, J., Evolutionary Modeling of Systems of Ordinary Differential Equations with Genetic Programming, *Genetic Programming and Evolvable Machines*, vol.1, pp.309-337, 2000.

[Iba94] Iba, H., deGaris, H., Sato, T., Genetic Programming using a Minimum Descrip-

Noise	0%		5%		10%	
	MSE	Success ratio	MSE	Success ratio	MSE	Success ratio
Our method	2.54^{-03}	100%	2.83^{-03}	100%	8.04^{-03}	67%
Standard GP	1.25^{-02}	40%	1.44^{-02}	33%	1.52^{-02}	33%

Table 3: The comparison of two methods in noisy environments.

tion Principle, in *Advances in Genetic Programming*, Kinnear,K.Jr. (ed.), MIT Press, pp.265-284, 1994.

[Koza92] Koza,J., Genetic Programming, MIT Press, 1992.

[Mimura and Iba02] Mimura,A. and Iba,H., Inference of a Gene Regulatory Network by Means of Interactive Evolutionary Computing, in Proc. of *Fourth Conference on Computational Biology and Genome Informatics*, (CBGI-02), 2002.

[Nikolaev and Iba01] Nikolaev,N. and Iba,H., Regularization Approach to Inductive Genetic Programming, *IEEE Transaction on Evolutionary Computation*, vol.5, no.4, 2001.

[Sakamoto and Iba00] Sakamoto, E., Iba, H., Identifying Gene Regulatory Network as Differential Equation by Genetic Programming, in Proc. of *11th Genome Informatics Workshop*, University Academy Press, 2000.

[Sakamoto and Iba01] Sakamoto, E., Iba, H., Inferring a System of Differential Equations for a Gene Regulatory Network by using Genetic Programming, in Proc. of *the 2001 Congress on Evolutionary Computation (CEC2001)*, 2001.

[Savageau76] Savageau, M.A., Biochemical Systems analysis: a study of function and design in molecular biology, Addison-Wesley, Reading, 1976.

by Y Takeuchi

[Takeuchi96] Takeuchi, Y., Global Dynamical Properties of Lotka-Volterra Systems ,World Scientific, 1996.

[Tominaga00] Tominaga, D., Koga, N., Okamoto, M., Efficient Numerical Optimization Algorithm Based
on Genetic Algorithm for Inverse Problem, Proc. of *Genetic and Evolutionary Computation Conference (GECCO2000)*, pp.251-258, 2000.

[Tomita99] Tomita,M., Hashimoto,K., Takahashi,K., Shimizu,T.S., Matsuzaki,Y., Miyoshi,F., Saito,K., Tanida,S., Yugi,K., Venter,J.C., and Hutchison,C.A.,3rd, E-CELL: software environment for whole-cell simulation, *Bioinformatics,* vol.15,no.1, pp.72-84, 1999.

[Zhang95] Zhang, B.-T. and Muehlenbein, H., Balancing accuracy and parsimony in genetic programming, *Evolutionary Computation*, vol.3, no.1, pp.17–38, 1995.

ABSTENTION REDUCES ERRORS —DECISION ABSTAINING N-VERSION GENETIC PROGRAMMING

Kosuke Imamura Robert B. Heckendorn Terence Soule James A. Foster

Initiative for Bioinformatics and Evolutionary STudies (IBEST),
Dept. of Computer Science,
University of Idaho, Moscow, ID 83844-1010
{kosuke,heckendo,tsoule,foster}@cs.uidaho.edu

Abstract

Optimal fault masking N-Version Genetic Programming (NVGP) is a technique for building fault tolerant software via ensemble of automatically generated modules in such a way as to maximize their collective fault masking ability. Decision Abstaining N-Version Genetic Programming is NVGP that abstains from decision-making, when there is no decisive vote among the modules to make a decision. A special course of action may be taken for an abstained instance. We found that decision abstention contributed to error reduction in our experimental *Escherichia coli* DNA promoter sequence classification problem. Though decision abstention may reduce errors, high abstention rate makes the system of little use. This paper investigates the trade-off between abstention rate and error reduction.

1 INTRODUCTION

This paper investigates the effect of an abstention threshold on the trade-off between abstention rate and error reduction, using an N-Version Genetic Programming ensemble classifier [1].

An ensemble binary classifier makes a yes/no decision based on votes from the participating ensemble member classifiers. A decision abstention occurs, when there is no decisive vote among the ensemble modules to make decision. An unanimous vote is the most decisive (highest ensemble confidence), while a tie vote is the least decisive (lowest ensemble confidence). The abstention threshold is set somewhere between these two extremes. If the vote count of either "yes" or "no" does not reach to this threshold, the ensemble abstains from decision-making. The ensemble, thus, produces three outputs: *yes*, *no*, and *don't know*. A special course of action may be taken for an abstained instance (such as classification by human experts) [2]. Abstention reduces the number of errors, potentially avoiding overfitting [2]. However, if the ensemble classifier abstains too often, it is of little use. Our experimental test problem is *Escherichia coli* DNA promoter sequence classification. This problem has been explored with artificial neural networks [3][4][5] and genetic programming [6].

1.1 BRIEF INTRODUCTION OF N-VERSION GENETIC PROGRAMMING (NVGP)

N-Version Genetic Programming (NVGP), which provides an optimal fault masking ensemble of automatically generated modules, is a new technique for building fault tolerant software that significantly reduces errors when applied to an *E. coli* promoter sequence classification problem [1]. Genetic programming is used to provide a large pool of candidate modules with sufficient diversity to allow us to select an ensemble whose faults are nearly uncorrelated. We find ensembles with a high degree of fault masking by randomly sampling from this large pool of modules. The ensembles that produce the expected error rate are retained. The expected failure rate f for n independent components, each of which fails with probability p, where the composite system requires m component faults in order to fail (initially derived for n-modular redundant hardware systems [7]) is

$$f = \sum_{k=m}^{n} \binom{n}{k} \left((1-p)^{n-k} \, p^{k} \right).$$

For an N-version classifier system, such as ours, the i-th individual fault rate p_i is the ratio of misclassified examples to the total number of training instances. In this case, f is the failure rate of an ideal ensemble. If the fault rate is the same for every p_i, f is an area under a binomial probability density function as shown in the above formula. The failure rate of an ensemble is close to the theoretically optimal rate f precisely when component failures are not correlated. This is our criterion for selecting the *qualified* ensembles. Explicit quantification of the module diversity with the theoretical failure probability is a distinct feature of NVGP, guaranteeing phenotypic diversity and the optimal ensemble.

N-version programming (NVP) was an early approach to building fault tolerant software that adapted proven hardware approaches to fault tolerance [8]. A fundamental assumption of the NVP approach was that independent programming efforts would reduce the probability that similar errors will occur in two or more versions [9]. But this assumption was questioned, because experiments showed that modules developed for NVP tended to fail under similar circumstances. For example, Knight and Leveson rejected the hypothesis of the assumed independence of faults by independently developed programs [10]. However, this conclusion does not invalidate NVP in general. Hatton showed that his 3-version NVP system increased the reliability by a factor of 45. Though this is far less than the theoretical improvement of a factor of 833.6, it is still a significant improvement in system reliability [11].

The next section reviews previous work on abstention and ensemble methods.

2 PREVIOUS WORK

Different ensemble construction methods have been studied in an effort to enhance accuracy. This section reviews abstention, averaging, median selection, boosting, and bagging. All methods exploit heterogeneity of ensemble components in one way or another.

2.1 ABSTENTION

Freund, et al., showed the error bound of averaging classifier with abstention [2]. The abstention threshold is based on the log ratio of the weighted sum of positive predictions and negative predictions. If the absolute value of this log ratio is smaller than the threshold, the ensemble classifier abstains from predicting. They identify the region of abstention as the locations of potential overfitting. Their theoretical work shows that the error of such predictor cannot be worse than twice of the best individual.

2.2 AVERAGE AND MEDIAN

A simple averaging method gathers outputs from all component modules and calculates their arithmetic average. Imamura and Foster showed simple averaging reduces error margins in path prediction [12] and function approximation with evolved digital circuits [13]. Another approach is weighted averaging, in which component modules are assigned optimal weights for computing a weighted average of the module outputs. Linearly optimal combination of artificial neural networks takes this approach [14][15]. Zang and Joung proposed Mixing Genetic Programs (MGP). MGP chooses a pool of individuals from a population and the master unit assigns the voting weights to these individuals using an additive weighting scheme [16]. The median value of the outputs is then the ensemble output. Soule approximated the sine function by taking the median of individuals, which were evolved, with subset of the entire training set for specialization [17]. Brameier and Banzhaf evolved teams of predictors. The individuals are coevolved as a team as opposed to post-evolutionary combination [18].

2.3 BOOSTING AND BAGGING

Boosting and bagging are methods that perturb the training data by resampling to induce classifier diversity. The `AdaBoost` algorithm trains a weak learner (slightly better than random guessing) by iterating training while increasing the weights of misclassified samples and decreasing the weights of correctly classified ones [19]. The effect is that the weak learner focuses more and more on the misclassified samples. The trained classifiers in each successive round are weighted according to their performance. The final decision is a weighted majority vote. Bagging (Bootstrap aggregating) replicates training subsets by sampling with replacement [20]. It then trains classifiers separately on these subsets and builds an ensemble by aggregating these individual classifiers. For evolutionary computation, Iba applied Boosting and Bagging to genetic programming and his experiment validated their effectiveness and their potential for controlling bloat [21]. Land used a boosting technique to improve performance of Evolutionary Programming derived neural network architectures in a breast cancer diagnostic application [22]. However, both techniques have limitations. Boosting is susceptible to noise, Bagging is not any better than a simple ensemble in some cases, neither Boosting nor Bagging is appropriate for data poor cases, and bootstrap methods can have a large bias [19, 23, 24, 25, 26, 27].

2.4 CLASSIFICATION OF ENSEMBLES

Table 1 categorizes current ensemble methods in genetic programming in terms of their sampling technique in combination with the evolutionary approach. In cooperative methods [17][28], speciation pressure (such as that caused by crowding penalties [28]) plays a vital role in evolving heterogeneous individuals, while in isolation methods there is no interaction between individuals during evolution. Resampling methods create different classifiers by using different training sets (bagging) or varying weights of training instances (boosting). Non-resampling methods create different classifiers from the same training set with or without explicit speciation pressure. NVGP and Decision abstaining NVGP are non-resampling techniques based on isolated evolution of diverse individuals.

Table 1. Classification of ensemble creation methods.

Evolutionary	**Training set selection**	
Approach	**Resampling**	**Non-resampling**
Non-Isolation	Boosting	Crowding
Isolation	Bagging	NVGP

3 EXPERIMENT

This section briefly summarizes the NVGP computational method (for detail description, see [1]) and presents the test results. We first run NVGP then incorporate abstention threshold to it. Our experimental problem is to classify whether a given DNA sequence is an *E. coli* promoter, using a decision abstaining NVGP. The data set is taken from UCI ML repository [29]. It contains 53 *E. coli* DNA promoter sequences and 53 non-promoter sequences of length 68.

3.1 COMPUTING ENVIRONMENT

The cluster supercomputing facilities from the Initiative for Bioinformatics and Evolutionary STudies (IBEST) is used to implement distributed computation. This device uses commodity computing parts to build substantial computing power for considerably less money than traditional supercomputers[1]. (http: //www.cs.uidaho.edu/ thecollective). This machine enabled experiments that would normally run for a month to complete in half a day.

3.2 INPUT AND OUTPUT

We used 2-gram encoding for input [30]. The 2-gram encoding counts the occurrences of two consecutive input characters (nucleotides) in a sliding window. Since there are four characters in DNA sequences ("a", "c", "g", "t"), we have 16 unique two-character strings to count. For example, a sequence "caaag" will be encoded as {ca=1, aa=2, ag=1}. The classifier clusters the positive instances and places the negative instances outside the cluster. The cluster is defined by the mean output value of postitve instances ± 3*(standard deviation). If an output value from a given sequence falls in the cluster, it is classified as a promoter. Otherwise, it is classified as a non-promoter.

3.3 CLASSIFIER

3.3.1 Target Machine Architecture

Our classifier is a linear genome machine [31], which mimics MIPS architecture [32]. There are two instruction formats in this architecture: `(Opcode r1,r2,r3)` and `(Opcode r1,r2,data)`. The instructions are ADDI, ADDR, MUL, DIV, MULI, DIVI, SIN, COS, LOG, EXP, NOP, MOVE, LOAD, CJMP, and CJMPI. The length of an individual program is restricted to a maximum of 80 instructions. Each evolving individual (a potential component for our NVGP ensemble system) used sixteen read-only registers for input data, which contained counts for individual nucleotide 2-grams as described above, and four read/write working registers.

[1] The total cost of the machine is about US$44,000. Micron Technology generously donated all of the memory for the machine.

3.3.2 Genetic Programming

We used 5 crossover methods. Methods (1) and (2) are traditional one and two point crossover, respectively. Method (3) is one point crossover with inversion applied to each crossover segment. Methods (4) and (5) use four random crossover points, with (5) being a single parent recombination operator. Fitness is calculated by the following correlation formula

$$C = \frac{PN - P_f N_f}{\sqrt{(N+N_f)(N+P_f)(P+N_f)(P+P_f)}}$$

where P and N are numbers of correctly identified positives and negatives, and P_f and N_f are the numbers of falsely identified positives and negatives [33]. Steady state is used for population replacement. Evolution continues until an individual of fitness 0.8 or above appears.

3.3.3 Evolution and Ensemble Testing

A common holdout test divides the dataset into 2 exclusive sets, 2/3 for the training set and 1/3 for the test set [27]. Our training sets used a random sample of 35 (53*2/3) positive and 35 negative examples, and used the remaining examples for the test sets. We performed experiments for 10 different holdout sets. The evolution and ensemble procedures are described below:

1. Create a training set and test set.
2. Evolve 40 isolated islands with 100 individuals each in parallel. Add an individual whose fitness is 0.8 from each island to a set B of single best versions.
3. Select N individuals by uniform-random sampling from B for N=15, 31 to form an NVGP ensemble. See 3.4.1 for the sampling frequency.
4. Evaluate the performance of each ensemble. If the ensemble is *qualified*, then retain it for a test set trial. Goto 3. The ensemble is *qualified* if the difference between the number of errors expected when versions have independent faults and the number of errors observed is small (less than one in our case).

3.4 EXPERIMENTAL RESULTS

The evolution and ensemble testing procedure described in section 3.3 is repeated for 10 different holdout tests in an attempt to reduce stochastic errors caused by sampling in performance estimation. We first show the performance of NVGP without abstention, then with abstention. We assume the number of errors have a normal distribution, since each test instance can be viewed as a Bernoulli trial [27].

3.4.1 Performance of NVGP

There are 40×10^9 and 27×10^7 possible ensembles to be formed respectively for 15 and 31 voter systems out of 40 candidate modules. Uniform random search sampled approximately 40×10^3 and 27×10^3 ensembles for 15 and 31 voter ensembles respectively, from which we selected

qualified ensembles for statistics. Table 2 shows the numbers of qualified ensembles found for each test. For example, we found 23199 qualified 15-voter ensembles out of 40×10^3 samples for the test 1. Table 3 is the result of t-test on the null hypothesis that average performance of the ensembles and the single best versions is not significantly different. Table 4 is the result of F-test on the null hypothesis that standard error of the ensembles and the single best versions is not significantly different. Table 5 shows error reduction percentage observed in ensembles relative to the error rates of the single best versions in the set B (see 3.3.3). It represents the average error reduction achieved by NVGP over single modules produced by genetic programming.

Figure 1 presents the performance distribution intervals of the single best versions and the corresponding *N*-voter NVGP ensemble at a 90% limit. For each holdout test, we present statistics for the single best versions, and for each of the four NVGP ensembles (N=15, 31). For example, the leftmost bar in holdout test 1 is the performance distribution of the 40 single best versions, showing that the best is estimated to be 20% error and the worst to be 48%, with a mean of 34%. The middle bar is 15-voter and the rightmost bar is 31-voter ensembles.

Table 2. The number of sampled qualified-ensembles

	test1	*test2*	*test3*	*test4*	*test5*	*test6*	*test7*	*test8*	*test9*	*test10*
15-voter	23199	29370	11596	8601	15973	4267	30455	32141	13171	17279
31-voter	19650	27205	9910	3197	13778	814	27340	27340	7672	23113

Table 3. The result of t-test, degree of freedom .≅ 40 for all the test cases.

	test1	*test2*	*test3*	*test4*	*test5*	*test6*	*test7*	*test7*	*test9*	*test10*
15-voter	11.07	9.10	3.27	8.67	9.67	10.21	8.80	6.06	6.64	5.46
31-voter	14.08	10.47	2.14	8.17	10.54	13.53	9.30	5.78	6.62	6.94

Table 4. The result of F test on error rate standard deviations

	test1	*test2*	*test3*	*test4*	*test5*	*test6*	*test7*	*test7*	*test9*	*test10*
15voter	3.15	4.10	4.98	3.53	4.01	2.22	2.71	4.37	7.22	3.76
31-voter	7.17	6.47	12.03	9.36	10.89	4.71	8.13	24.11	47.44	8.07

Table 5. Percentage error reduction of NVGP relative to set of best individual in isolation

	test1	*test2*	*test3*	*test4*	*test5*	*test6*	*test7*	*test7*	*test9*	*test10*
15-voter	44	33	14	31	40	31	37	22	25	24
31-voter	56	38	9	29	43	42	39	21	25	31

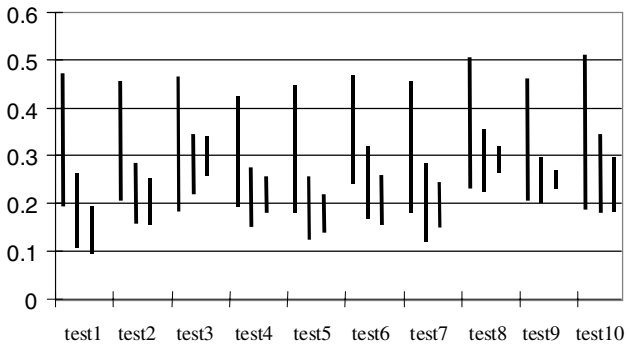


Figure 1. Error rate distribution intervals of the single best versions and the corresponding *N*-voter NVGP ensemble at a 90% limit. Leftmost, middle, and rightmost bars are distribution of single-version, 15-voter, and 31 voter system respectively.

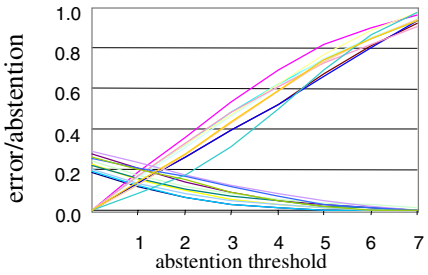


Figure 2. abstention rate and error rate for 15 voter system

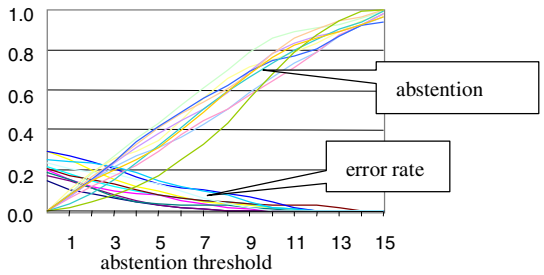


Figure 3. abstention rate and error rate for 31 voter system

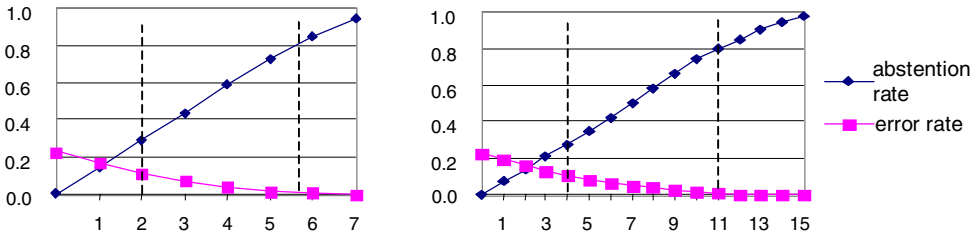


Figure 4. average abstention and error rates from figure 2 and 3.

3.4.2 Behavior of Decision Abstaining NVGP

The abstention thresholds are incorporated into the NVGP outputs of 3.4.1. The decision abstaining ensemble requires abstention threshold, h, and needs $((N+1)/2 + h)$ votes, either positive or negative, to make a decision, where N (odd) is the number of vote participating individuals. The voting scheme is a simple majority rule, if h=0. Figure 2 and 3 are the plots of the abstention rates and the error rates of 15 and 31 voter ensemble for the 10 holdout tests with respect to the abstention threshold, h. Figure 4 represents the average abstention and error rates from figure 2 and 3 for collective analysis. The error rate is a decreasing function and the abstention rate is an increasing function.

4 DISCUSSION

4.1 NVGP

Though a holdout test is commonly used to measure performance of evolutionary algorithms, it is not reliable. Kohavi argues that holdout testing does not provide a good estimate of error rate [27]. Nonetheless, we repeated the holdout test 10 times with different training/test sets for somewhat fair statistics. In figure 1, the hold-out test 3 does not exhibit apparent superiority of NVGP as in the test 1, though we reject the null hypothesis that average performance of single best version and NVGP are not significantly different at α=0.975. For all the other nine test cases, we reject the hypothesis virtually at 100% and conclude that NVGP is superior. NVGP error rates in all ten tests are far below the theoretical bound shown by Freund [2] even without abstention. Table 4 indicates that performance fluctuation of NVGP is statistically significantly smaller than single versions. Apparently, as the ensemble size approaches to the pool size, the performance fluctuation becomes smaller. If we combine all the individuals in the pool, there is no performance fluctuation. Therefore, a larger fluctuation may be expected for NVGP if the component pool size is huge. But, also true is that duplicate phenotypes start populating the pool as the pool size becomes larger. In fact, our experiment witnessed that an exhaustive search for an optimal ensemble of 39 voters from the pool failed in three out of the ten holdout tests. This possibly indicates that the entropy of the pool may have reached a plateau with the given training data and training method. If this is the case, the small performance fluctuation for *optimally sized* NVGP will still hold regardless of the pool size increase. Further study is needed for an optimal size of NVGP.

Notice that a single best individual has a chance to become practically a random classifier (error rate above 0.4) roughly 10%-20% of the time on unseen data. Unfortunately, we have no way of knowing which individual would become a random classifier beforehand, because they all have the same fitness (0.8) on the training set. This is the risk we must bear with a single best classifier. Fluctuation in performance is the very reason why we compared the distributions, and why NVGP has superior performance.

4.2 ABSTENTION

Figure 4 shows (see dashed lines) that the decision abstaining NVGP achieved a near zero error rate, at high

Table 6. Q values for 31-voter ensembles (ρ=0.5)

abstention threshold	*test1*	*test2*	*test3*	*test4*	*test5*	*test6*	*test7*	*test8*	*test9*	*test10*
0	6.7	8.0	10.1	7.7	6.8	8.8	7.3	10.4	9.1	9.5
1	7.2	8.3	10.0	7.9	7.1	9.1	7.6	10.6	9.0	9.7
2	8.5	9.2	9.8	8.4	7.8	9.9	8.6	11.1	9.1	10.2
3	10.5	10.5	10.1	9.5	9.3	11.2	10.1	12.2	9.8	11.0

cost of abstention rate, approximately 80%, for both 15-voter and 31-voter ensembles. Abstention rates 29% of 15-voter and 28% of 31-voter ensembles give 50% error reduction over NVGP alone (no abstention). Whether these abstention rates are acceptable for error reduction depends on how critical it is to have wrong predictions.

The abstention rates and the error rates are monotonic with respect to the abstention thresholds, and the trade-off between abstention and error reduction can be estimated almost linearly. Consequently, there is no analytically measurable peak gain by abstention. Subjective judgment must be used to set the abstention threshold. The following formula may be used to numerically measure the effect of abstention: $Q= E_a + \rho N$, where E_a is the number of errors with abstention, N is the number of *don't know* outputs, and $0 \le \rho \le 1$. If ρ=1, then *don't know* is as bad as wrong prediction and counted as an error. On the other extreme, if ρ=0, it is as good as correct prediction. The larger the ρ value is the more penalties for *don't know* outputs.

Let the number of errors of NVGP alone (no abstention) be E_z. If $Q \le E_z$, then we are unconditionally better off with abstention. For example, setting ρ=0.5 (half way between correct and incorrect prediction), we obtain Q values for 31-voter ensembles as shown in Table 6. The Q values are fairly close to E_z when the threshold is 1, which gives 3.2% error reduction (Figure 4 data). In other words, threshold = 1 is a break-even point for the trade-off between abstention and error reduction for ρ=0.5. For safety critical applications, such as medical diagnostics, a smaller ρ value would be appropriate for the trade-off analysis. That is to say, do not penalize heavily when an ensemble is trying to avoid a random guess. It may well be the case where the training set was inappropriate for particular instances.

4.3 POST-EVOLUTIONARY COMBINATION

Post-evolutionary combination is thought to be computationally inefficient, because many runs are required to obtain a sufficient number of individuals [18]. However, inexpensive cluster computing alleviates this problem (see section 3.1). Not only can the post-evolutionary search for the optimal NVGP ensemble be performed in parallel, but also the search may no longer need to be continued after an optimal ensemble is found.

5 CONCLUSION AND FUTURE RESEARCH

We showed the experimental classification result by NVGP, which significantly improved accuracy and reduced the performance fluctuation. Then, we incorporated decision abstention to it. Abstention in effect avoids random guesses when the ensemble confidence is low, i.e., votes are too close to call. It is a viable method to reduce errors. The trade-off between abstention and error reduction is subjective. The abstention threshold value depends on how critical an application is.

It is important to curve the abstention rate increase. We plan to embed the individual confidence to enhance the ensemble confidence. The individual confidence, in our case, can be measured by the distance of an instance from the cluster center. The further the distance, the lower the confidence. The ensemble confidence in prediction is measured by the level of disagreement among the voters.

Acknowledgments

This work is supported by the Initiative for Bioinformatics and Evolutionary STudies (IBEST) at the University of Idaho; by NIH NCRR grant 1P20RR016454-01; and by NIH NCRR grant NIH NCRR 1P20RR016448-01; and by NSF grant NSF EPS 809935. We are grateful to colleagues in the Initiative for Bioinformatics and Evolutionary STudies (IBEST) for insightful discussions. Finally, we wish to thank the many students who designed and built our Beowulf cluster, and Micron Technology for donating the memory for that machine. We thank John Cavalieri and Janet Holmberg for proofreading.

References

1. Imamura, K., Heckendorn,R B., Soule, T., Foster, J A.: N-version Genetic Programming via Fault Masking Proceedings of the Euro EuroGP2002 5th European Conference on Genetic Programming (To appear)
2. Freund, Y., Mansour, Y., Schapire, R E.: Why Averaging Classifiers Can Protect Against Overfitting. Proceedings of the 8th International Workshop on Artificial Intelligence and Statistics,2001 (www.ai.mit.edu/confereces/aistats2001/papers.html)

3. Pedersen, A.G., Engelbrecht, J.: Investigations of *Escherichia Coli* promoter sequences with artificial neural networks: New signals discovered upstream of the transcriptional startpoint. Proceedings of the Third International Conference on Intelligent Systems for Molecular Biology (1995) 292-299 (http://citeseer.nj.nec.com/25393.html)
4. Towell,G.G., Shavlik, J.W., Noordewier, M.O.: Refinement of approximate domain theories by knowledge-based neural networks. Proceedings of AAAI-90 (1990) 861-866 (http://citeseer.nj.nec.com/towell90refinement.html)
5. Ma, Q., Wang, J.T.L.: Recognizing Promoters in DNA Using Bayesian Neural Networks. Proceedings of the IASTED International Conference, Artificial Intelligence and Soft Computing (1999) 301-305 (http://citeseer.nj.nec.com/174424.html)
6. Handley, S.: Predicting Whether Or Not a Nucleic Acid Sequence is an *E. Coli* Promoter Region Using Genetic Programming. Proceedings of First International Symposium on Intelligence in Neural and Biological Systems, IEEE Computer Society Press, (1995) 122-127
7. Pradhan, D. K., Banerjee, P.: Fault-Tolerance Multiprocessor and Distributed Systems: Principles. In Pradhan, D.K.: Fault-Tolerant Computer System Design. Chapter 3, Prentice Hall PTR, (1996), 142
8. Avizienis, A. and J.P.J. Kelly: Fault Tolerance by Design Diversity: Concepts and Experiments. IEEE Computer, vol. 17 no. 8, (1984), 67-80
9. Victoria Hilford., Lyu, M. R., Cukic B., Jamoussi A., Bastani F. B.: Diversity in the Software Development Process. Proceedings of Third International Workshop on Object-Oriented Real-Time Dependable Systems, IEEE Comput. Soc, (1997), 129-36 (http://www.cse.cuhk.edu.hk/~lyu/papers.html#SFT_Techniques)
10. Knight, J.C., Leveson, N.B.: An Experimental Evaluation of the Assumption of Independence in Multiversion Programming. IEEE Transaction on Software Engineering, vol. SE-12, no. 1 (1986)
11. Hatton, L.: N-version vs. one good program. IEEE Software, vol 14, no. 6 (1997) 71-76
12. Imamura, K., Foster, J.A.: Fault Tolerant Computing with N-Version Genetic Programming. Proceedings of Genetic and Evolvable Computing Conference (GECCO), Morgan Kaufmann, (2001) 178
13. Imamura, K., Foster, J.A.: Fault Tolerant Evolvable Hardware Through N-Version Genetic Programming. Proceedings of World Multiconference on Systemics, Cybernetics, and Informatics (SCI), vol. 3, (2001) 182-186
14. Hashem, S.: Optimal Linear Combinations of Neural Networks. Neural_Networks, vol. 10, no. 4, (1997) 599-614 (http://www.emsl.pnl.gov:2080/proj/neuron/papers/hashem.nn97.abs.html)
15. Hashem, S.: Improving Model Accuracy Using Optimal Linear Combinations of Trained Neural Networks. IEEE Transactions on Neural Networks, vol.6, no.3 (1995) 792-794 (www.emsl.pnl.gov :2080/proj/neuron//papers/hashem.tonn95.abs.html)
16. Zang, B-T., Joung, J-G.: Enhancing Robustness of Genetic Programming at the Species Level. Proceedings of the 2nd Annual Conference Genetic Programming 97, Morgan Kaufmann (1997) 336-342.
17. Terence Soule, "Heterogeneity and Specialization in Evolving Teams", Proceeding of Genetic and Evolvable Computing Conference (GECCO), Morgan Kaufmann (2000) 778-785
18. Brameier, M., Banzhaf, W.:Evolving Teams of Predictors with Linear Genetic Programming. Genetic Programmin and Evolvable Machines, vol. 2, (2001) 381-407
19. Schapire R.E., Freund, F.: A Short Introduction to Boosting. Journal of Japanese Society for Artificial Intelligence 14, no. 5, (1999) 771-80 (http://citeseer.nj.nec.com/freund99short.html)
20. Breiman, L.: Bagging Predictor. Technical Report No.421, Department of Statistics, University of California Berkley, 1994 (http://www.salford-systems.com/docs/BAGGING_PREDICTORS.PDF)
21. Iba, H.: Bagging, Boosting, and Bloating in Genetic Programming. Proceedings of the Genetic and Evolutionary Computation Conference, vol. 2, Morgan Kaufmann, (1999) 1053-1060
22. Land, W.H. Jr., Masters T., Lo J.Y., McKee, D.W., Anderson, F.R.: New results in breast cancer classification obtained from an evolutionary computation/adaptive boosting hybrid using mammogram and history data. Proceedings of the 2001 IEEE Mountain Workshop on Soft Computing in Industrial Applications. IEEE, (2001) 47-52
23. Basak, S.C., Gute, B.D., Grunwald, G.D., David W. Opitz, D.W., Balasubramanian, K.: Use of statistical and neural net methods in predicting toxicity of chemicals: A hierarchical QSAR approach. Predictive Toxicology of Chemicals: Experiences and Impact of AI Tools - Papers from the 1999 AAAI Symposium, AAAI Press, (1999) 108-111
24. Opitz, D.W., Basak, S.C., Gute, B.D.: Hazard Assessment Modeling: An Evolutionary Ensemble Approach. Proceedings of the Genetic and Evolutionary Computation Conference, vol. 2, Morgan Kaufmann (1999) 1643-1650
25. Maclin, R., Opitz, D.: An empirical evaluation of bagging and boosting. Proceedings of the Fourteenth International Conference on Artificial Intelligence, AAAI Press/MIT Press (1999) 546-551 (http://citeseer.nj.nec.com/maclin97empirical.html)

26. Bauer, E., Kohavi, R.: An Empirical Comparison of Voting Classification Algorithms: Bagging, Boosting, and Variants. Machine Learning, vol. 36, 1/2, Kluwer Academic Publishers (1999) 105-139 (http://citeseer.nj.nec.com/bauer98empirical.html)

27. Kohavi, R.: A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection. Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI), Morgan Kaufmann (1995) 1137-1145 (http://citeseer.nj.nec.com/kohavi95study.html)

28. Soule, T.: Voting Teams: A Cooperative Approach to Non-Typical Problems. Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-99), vol. 1,Morgan Kaufmann (1999) 916-922

29. UCI Machine Learning Repository, Molecular Biology Databases (http://www1.ics.uci.edu/~mlearn/MLSummary.html)

30. Wang, J.T.L., Ma, Q., Shash D., Wu, C.: Application of neural networks to biological data mining: a case study in protein sequence classification. Proceedings KDD-2000. Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, (2000) 305-309 (http://citeseer.nj.nec.com/382372.html)

31. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic Programming: An Introduction: On the Automatic Evolution of Computer Programs and Its Applications. Academic Press/Morgan Kaufmann (1998)

32. MIPS32™ Architecture for Programmers Volume I: Introduction to the MIPS32™ Architecture (http://www.mips.com/publications/index.html)

33. Matthwes, B. W.:Comparison of the predicted and observed secondary structure of T4 phage lysozyme. Biochimica et Biophysica Acta, vol. 405 (1975) 443-451

Collaborating with a Genetic Programming System to Generate Modular Robotic Code

Jeremy Kubica*
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15232
jkubica@ri.cmu.edu

Eleanor Rieffel
FXPAL
3400 Hillview Ave Bldg 4
Palo Alto CA 94304
rieffel@fxpal.com
(650) 813-7077

Abstract

While the choice of a set of primitives can strongly effect the performance of genetic programming, the design of such a set remains more of an art than a science. We look at a joint approach that combines both hand-coding and genetic programming to define and refine primitives in the context of a creating distribute control code for modular robots. We give some rules of thumb for designing and refining sets of primitives, illustrating the rules with lessons we learned in the course of solving several problems in control code for modular robots.

1 INTRODUCTION

It is well-known, from various "no free lunch" theorems, that no single approach can effectively solve all problems (Wolpert, 1996; Wolpert and Mecready, 1997). Thus it is necessary to tailor techniques to the problem at hand. For genetic programming this tailoring can be done in three primary ways: choosing the primitives, choosing the fitness functions, and choosing parameters for the algorithm such as the percentage of use of various operators. Here we look at the problem of effectively choosing primitives.

We describe techniques for and experiences with the joint use of genetic programming and hand-coding to design and refine sets of primitives for a variety of modular robotics problems. The creation of decentralized control software for modular robots is a difficult problem due to both the decentralized nature of the software and the fact that the connectivity relations between the modules constantly change. Yet, in spite of these difficulties, or perhaps because of them, modular robotics provides an ideal domain in which to experiment with automated software generation methods; there are many robotic tasks that are easily specified but for which it is highly non-obvious what distributed software would create the desired behavior.

Ideally, for the problem of robotic control, a set of primitives could be derived directly from a description of the hardware capabilities. While a description of such capabilities is an excellent place to start, finding a solution given only the most basic actions can be prohibitively time consuming. Another approach is to tailor the primitives to the task, which may require significant development time. In fact, the development time required for the primitives can approach that of actually solving the problem. We take an intermediate approach. Except for primitives that encode information specific to the problem (as opposed to its solution), we try to provide primitives that would appear to be useful in a wide range of problems and are still reasonably low-level. We use insights gained from both human and machine attempts to solve the problems to design effective sets of primitives.

2 MODULAR ROBOTIC HARDWARE AND SIMULATOR

2.1 TELECUBE MODULES

Modular self-reconfigurable robots are systems consisting of a collection of simple and identical robotic modules that can form connections to and move relative to each other (Yim, 1994; Murata et. al., 1994; Rus and Vona, 2000). These modules function together to produce an overall behavior of the robot, analogous to the way cells in a body function together. The fact that both the connectivity and relative positions of the modules can change allows the overall robot to reconfigure and take on different shapes. Modular robotic

* Supported by FXPAL

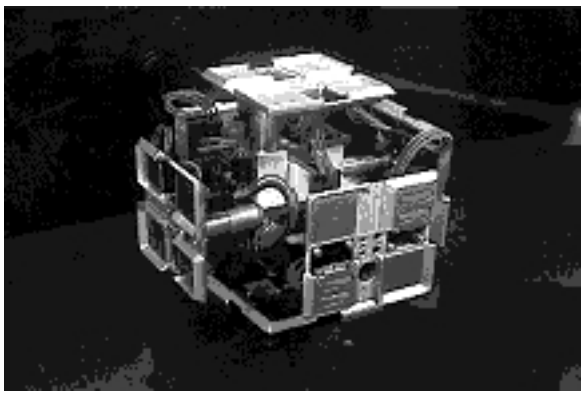


Figure 1: One telecube module.

systems provide possible benefits in terms of flexibility, adaptability, and robustness.

2.2 MODULE CAPABILITIES

2.2.1 Physical Capabilities

The robot modules we consider are the TeleCube modules currently being developed at Xerox PARC and shown in Figure 1 (Suh et. al., 2002). They are similar in design to modules being designed and build at Dartmouth (Rus and Vona, 2000), which can expand in two dimensions. The modules are cube shaped with an extendable arm on all six faces. The arms are assumed to extend independently up to half of the body length, giving the robot modules an overall 2:1 expansion ratio. For simplicity, we restrict the arms to fully extended or fully retracted states. The expansion and contraction of these arms provide the modules with their only form of motion. Latches on the plates at the end of each arm allow two aligned modules to connect to each other. The arm motion together with the latching and unlatching capability means that the connectivity topology of a modular robot can change greatly over time. As shown in (Vassilvitskii et. al., 2002), this motion is sufficient to enable arbitrary reconfiguration within a large class of shapes.

2.2.2 Sensors, Communication, and Memory

Each module is assumed to have simple sensing and communication abilities that resemble those that will be given to the TeleCube modules currently being built at Xerox PARC. Modules can send limited bandwidth messages to their immediate neighbors. Each module is also assumed to be able to sense contact at the end of each arm, sense how far each of their arms is extended, determine whether they are connected to a neighboring module, and detect nearby, adjacent modules. Each module has a small memory capacity, which is initialized to all zeros. We also give the modules simple computational abilities such as the capability to determine the opposite of a given direction, the ability to generate a random direction, and to calculate and store the value of each of its position coordinates.

2.3 TELECUBE SIMULATOR

The experiments reported here were run by connecting FXPAL's genetic programming system to a simulator, written by J. Kubica and S. Vassilvitskii, for the TeleCube modular robots. The module control code is represented in a LISP-like parse tree that is evaluated once per time step for each module.

3 PROBLEM CONTEXTS

For all of the problems described here we are interested in a completely decentralized solution in which the desired global behavior of the robot emerges from control code that is run locally on each of the modules that make up the robot. Decentralized control software is a challenging domain to begin with, and the fact that the connectivity relations between the modules constantly change makes modular robotic problems even harder. The collaboration between hand-coding and automatic generation of solutions is described below in the context of three specific modular robotic control problems to which we have applied this technique: the tunnel problem, the filtering membrane problem, and the sorting membrane problem.

The *tunnel problem* consists of a long thin world, 40 x 10 x 2 arm lengths, that is enclosed on all sides by walls. During each of the trials an object is placed randomly "in" one of the long walls. This means that a module adjacent to this location along the wall can sense the object, but will not be blocked by it or stuck behind it while moving along the wall. The goal of the modules is to find the object and all move as close to it as possible. Thus, we define our fitness function as the sum of each module's distance to the object at the end of a run. The modules start out as a 3 x 3 x 1 grid configuration in one corner of the world.

The membrane problems consist of a membrane, a three-dimensional lattice of modules, and a foreign object which, depending on its attributes, should or should not be accepted into the membrane and manipulated by it. In the case of a *filtering membrane*, the object is either accepted or rejected. Accepted objects are passed through the membrane and out the bottom, whereas rejected objects remain on top. A *sorting membrane* accepts all objects, but sorts them

along some axis. We looked at the case of a binary sorting membrane, which moves an object towards one of two opposite ends depending on its value. For both membrane problems we use fitness functions that measure the distance of the object from where it ought to end up and a penalty for the membrane breaking apart. Since the modules are unable to directly grasp or pull the objects, all solutions to the membrane problem require the use of gravity, by having modules move out from under the objects and allowing them to fall through the membrane. For more details on the membrane problems, including some pictures, see (Kubica and Rieffel, 2002).

4 RELATED WORK

A number of researchers have looked at enabling a genetic programming system to add primitives on its own in the course of its runs. Approaches include ADFs (Koza, 1994), libraries (Angeline and Pollack, 1994), subroutines (Rosca and Ballard, 1996), and subtree encapsulation (Roberts et al., 2001). While such techniques might help with some of the problems we describe here, we are particularly interested in how to add primitives that the system would be unlikely to find for itself. Furthermore, there are many situations, perhaps the majority, in which one is most interested in solving the problem at hand in any way possible, rather than being concerned about the extent to which the solution was automatically attained. We hope that what we write here can help others to have more productive collaborations with a genetic programming system in order to more effectively solve practical problems of interest.

The problem of automatic code generation for modular robots has also seen interest recently. Kubica et. al. (Kubica et. al., 2001) hand-coded control programs for internal object manipulation with robots made of TeleCube modules. Bennett et. al. (Bennett et. al., 2001) used genetic programming to generate distributed control programs for modular robots consisting of sliding-style modules (Bennett and Rieffel, 2000; Pamecha et al., 1996). It is important to note however that this sliding-style module design enables movement with primitive operations directly suggested by the hardware. In particular, movement in that setting, unlike for the TeleCube modules, does not require explicit connection and disconnection actions. Thus movement in their case avoids some of the difficulties we faced when attempting to generate effective software for robots made from Telecube style modules.

5 DEVELOPING EFFECTIVE SETS OF PRIMITIVES

5.1 BASIC PRIMITIVES

Each of the basic capabilities of the modules can be captured by a primitive operation.

- Physical actions: `(ExtendArm direction)`, `(RetractArm direction)`, `(Connect direction)`, `(Disconnect direction)`
- Communication: `(SendMessage direction type value)`, `(GetMessage direction type)`
- Sensors: `(HasNeighbor direction)`, `(ReadSensorNeighborDist direction)`, `(ReadSensorObjectDist direction)`
- Memory: `(ReadReg index)`, `(SetReg index value)`
- Other: `(OppDirdirection direction)`, `(RandDir)`, and `(GetX)`, `(GetY)`, `(GetZ)`.

In addition to these module specific primitive, we allow the module control programs to use the following basic programming primitives: `(Add)`, `(Sub)`, `(If)`, `(ProgN)` `(LT)`, `(And)`, and `(Not)`, and the numeric constants `0.0`, `0.1`, `0.2`, `0.4`, `0.6`, `0.8`, and `1.0`.

The primitives described above all mirror basic hardware capabilities of Telecube modules. For more details on these primitives, see (Kubica and Rieffel, 2002). However these primitives do not form an optimal set of primitives for either a human or a genetic programming system to construct effective software for the tasks we described above. We look at how genetic programming experiments and hand-coding attempts together enabled the development of effective sets of primitives.

5.2 INITIAL CHOICE OF PRIMITIVES

Ask the system only to discover a solution, not aspects of the problem

One important consideration when choosing primitives is to give the system full information about the problem. Even researchers more concerned with automatically generating code than solving problems per se and who are resistent to giving the system any hints as to how to solve the problem, should feel comfortable giving the system enough information so that it does not have to guess the problem as well as the solution.

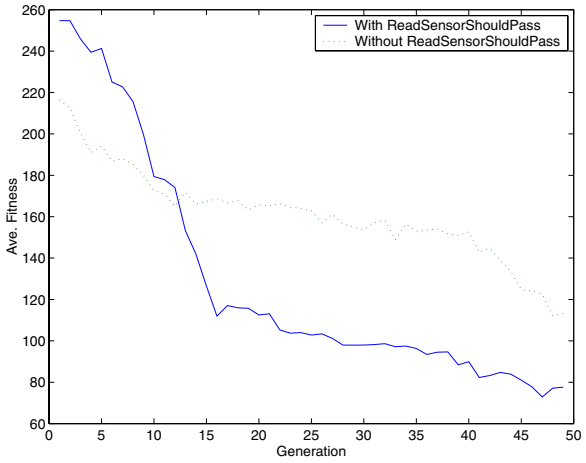


Figure 2: Average solution performance on filtering membrane problem with and without ReadSensorShouldPass primitive.

In the filtering membrane problem, the modules must determine whether or not to accept an object. In our case the criterion for the filter to accept an object was that the value returned by `(ReadObjectVal direction)` be above `0.5`. In order to solve the problem in our initial runs the GP system had to create a subtree testing for the acceptance criterion.

As we tried to gauge how the system was doing, we realized that we were using the existence of such a subtree to determine how close the GP system was to a solution. We soon realized that one should not require an automatic code generation technique to guess the problem as well as the solution, even if it can. Thus we added a primitive that encapsulates the filtering criterion: the Boolean `(ReadSensorShouldPass direction)` primitive

`(LT(Add(0.4 0.1))(ReadObjectVal direction))`.

Figure 2 shows the performance of two GP runs, one with the `ReadSensorShouldPass` primitive and one without it. While the system could solve the problem without the `(ReadSensorShouldPass direction)` primitive, its addition certainly helped. Further, for the evolution of the membrane control there is no reason to withhold this type of information about the problem statement itself. Note that this situation is different from one in which one might be trying to learn a good criterion, say for identifying defective parts or a distinguishing characteristic of a set of objects.

Avoid large needle-in-the-haystack searches

The tasks we describe above all require movement, as

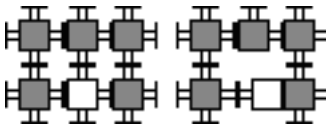


Figure 3: Module in white moves towards right.

do most modular robotic tasks including reconfiguration, locomotion, and internal manipulation. But none of the fitness functions we used give any reward to programs that have put together part, but not all, of a sequence that would result in movement. Since no reward is given, the genetic programming system has nothing to guide it towards movement.

Even at its simplest level, movement of a single TeleCube module is a complex process, involving bonding with and unbonding from neighbors, expanding and contract arms, and checking that the move is possible. To get an impression the complexity of a single movement, it is only necessary at look at the movement subtree involved in disconnecting from the perpendicular neighbors:

```
(ProgN (ProgN (Disconnect 0.0) (RetractArm
0.0)) (ProgN (Disconnect 0.2) (RetractArm
0.2)) (ProgN (Disconnect 0.4) (RetractArm
0.4)) (ProgN (Disconnect 0.6) (RetractArm
0.6)))
```

The code above disconnects from the correct neighbors in the case the module is moving along the $\pm$X axis. Just to choose `Disconnect` and `Retract` arm from just the four physical primitives has probablility less than 10^{-3}. Choosing the appropriate constants for the `Disconnect` and `Retract` functions from the seven numeric constants also has probability less than 10^{-5}. And choosing the five `ProbN` primitives has probability less than 10^{-5}. So the probability of finding the correct components is less than 10^{-13}.

Up to now we have only considered a subtree of the sort that would correctly disconnect a module. In addition, one would need appropriate `if` statements to determine the correct direction, the appropriate arm expansions and contractions to accomplish the movement, and a similar subtree to handle reconnection with any new neighbors. Thus, the simple move illustrated in Figure 3, given only the basic primitives described above, would require at least 12 different primitives and well over 50 nodes. More code would be required to enable moves in all six directions, not just one, and the graceful handling of movement failures would require considerably more code. Thus the likelihood that such a program would be created given

no guidance is impractically small.

With the basic primitives, and a simple fitness function with no gradient towards movement from none, the GP system would be searching blindly. Thus it is extrodinarily unlikely that the GP system would succeed even in evolving movement. In fact the problem is harder still since only useful and used motion would be rewarded. By asking genetic programming to evolve movement, we are asking it to search for a needle, or rather a few needles, in a very large haystack.

A number of solutions are possible. One solution would be to change the fitness function so that it rewards partial programs out of which movement could be constructed. Another possibility is to simply evolve movement first and then use that solution as a primitive when evolving solutions to more complex tasks. Note that this approach still requires determining a fitness function that rewards partial solutions. It is highly non-trivial to see how to design a principled reward system. Figuring out principled approaches to this problem, and other similar problems, is an open research issue that ought to be of great interest to anyone trying to automatically solve problems. An unprincipled way of solving this problem is to hand-code a solution and then reward programs based on similarity with this solution. One might as well use the hand-coded solution as a primitive in the first place, which is what we do.

Achieving simple motion, for instance the motion illustrated in Figure 3, only requires reasoning at the local level and is thus easier for a human to do than solutions to tasks that require global behavior to emerge from the local actions. Following basic operations used by Kubica, et. al. (Kubica et. al., 2001), we created a movement primitive that enables modules to move a distance of one arm length in any of the six directions. The movement is accomplished by simultaneously contracting the front arm and expanding the back arm to effectively "slide along the arms." The movement primitive also checks whether movement is possible, reverses any steps taken prior to a failed check, and returns whether or not the movement has succeeded.

This example also serves to illustrate another principle useful for effective collaboration with a GP system in solving a problem.

Hand-design parts that are easy for a human to write but hard for a GP system to discover.

5.3 REFINING THE PRIMITIVES

Delete or replace unused primitives

If the system is solving a problem without certain primitives consider removing or replacing them. Use this information to update your intuition if possible. Encapsulating such primitives is a particularly attractive choice in a number of situations.

Encapsulate hard-to-use primitives

If the system is not using a primitive, or a set of primitives, effectively, consider giving the system higher level primitives that it may be able to use more easily. The best time to encapsulate actions into primitives is when a certain subtree is needed.

The clearest case for removing a primitive is when a higher level primitive replaces it. For example, the implied TeleCube primitive `(Disconnect direction)` is automatically handled by such primitives as `(Move direction)` and `(RetractArm direction)`. Further, maintaining global connectivity is vital for power routing and facilitates alignment of connecting modules and inter-module communication. Thus, we wish to maintain a high level of connectivity at all times and additional uses of disconnection may not only be unnecessary, but also detrimental. Thus, it can be said that the disconnect primitive was replaced by incorporating it into higher-level primitives.

A second example of the removal of primitives is in the more complex problem of module communication. Since control is local, it would seem that communication between the modules would be required in order for the robot to achieve a task of any complexity. Messages would enable modules to share information and coordinate actions. However, communication may not be as necessary for modular robotic tasks as humans tend to think. Bennett and Rieffel comment in their conclusions that none of the solutions to any of the five tasks they studied (Bennett and Rieffel, 2000) used the communication capabilities provided. Similarly, while the solution given in (Bennett et. al., 2001) contains two `(SendMessage)` commands, since it contains no `(ReadMessage)` commands, one can deduce that the communication capabilities were not used.

The lack of use of the communication primitives in solutions to various problems suggest that a wider class of problems can be solved without communication than most humans would generally think. However, it also suggests that the provided communication primitives may be hard for a genetic programming system to use. Effective communication requires the generation of a quantity that would be useful to communicate, the sending of that quantity, the receipt of that quantity, and the use of that quantity by the receiver. In most cases, until all of these steps are in

place and correct there is no benefit. So for a genetic programming system, with a simple fitness function in which there are no explicit rewards for partial communication, the generation of effective communication is reduced to a large needle-in-the-haystack problem similar to the movement problem we described above. Unfortunately, in this case it is less clear what to do by hand since what sort of communication would be useful is less clear.

One problem with the message primitives described above is that it is difficult to send out information to all neighbors or send direction dependent information, such as "move away." One interesting extension to the communications capabilities described above is the addition of gradients. Gradients are messages that are broadcast to all surrounding neighbors. These neighbors in turn rebroadcast them to their neighbors, with the strength of the gradient decayed with each broadcast.

Previous work with gradient messages in hand-coded solutions was shown effective in (Bojinov et. al., 2000; Kubica et. al., 2001) and in (Shen et. al., 2000; Shen et. al. 2000) where the gradients were called scents and hormones respectively. Of particularly interest is the use of gradients for internal manipulation of objects. The use of scents applied to the problem of internal object manipulation in (Kubica et. al., 2001) limited the scents to positive and negative, where modules were inclined to move towards the positive scent and away from the negative scent. This implies further specializing the messaging primitives to:

- `(SendPositiveGradient)` Emits a positive gradient
- `(SendNegativeGradient)` Emits a negative gradient
- `(HandleGradient)` Tries to follow the gradients, move towards positive gradient and away from negative gradient. Returns `true` if a gradient was detected and the module was able to move to follow it.

These communication primitives illustrate how human experience and intuition can be encoded in general primitives. These primitives have the potential to greatly reduce the space of programs that might be searched in order to find a solution. For example, the positive and negative gradients above do not require the use of constants, simplifying the coordination and handling of messages.

The tunnel world problem was designed as a test of the communication primitives. It is relatively easy for the modules to move randomly around the world in a group and for a single module to stop moving if it detects an object. The problem gets increasingly non-trivial when all of the modules are required to move as close as possible to the object. This implies the need for at least implicit communication between the modules. Specifically, when a module detects the object it must be able to say "move towards me" or something similar.

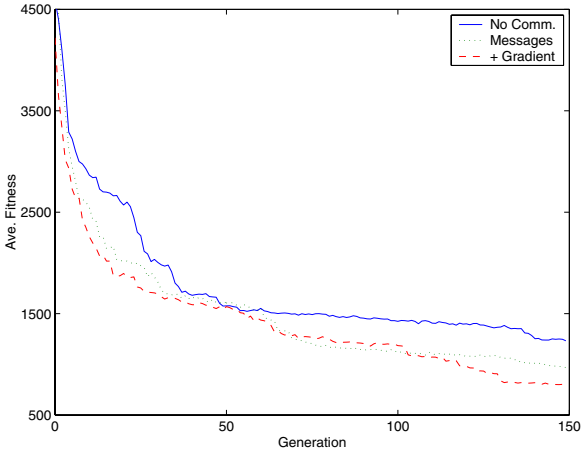


Figure 4: Average solution performance on the tunnel problem for runs with different sets of communication primitives.

To determine the effectiveness of various communication primitives, the problem was run 24 times in each of three conditions: 1) no communication primitives, 2) only the messaging primitives, and 3) only the positive gradient primitives (`SendPositive` and `HandleGradients`). The average cost for these trials during each generation of the GP runs was recorded and is shown in Figure 4. Once again, the intuitive difference is reflected in the success of the runs that were able to use the positive gradients. The solutions that can contain positive gradients perform the best after the 150 generations, while the solutions that could not have any message passing commands in them perform noticably worse.

Consider deleting unused primitives

A case can be made for keeping primitives that immediately appear to have no benefit. The argument goes that if they are not needed, GP will evolve solutions that do not use them. Despite this argument, it is important to place some restrictions on the number of primitives. As the number of primitives increases, the search space of possible programs increases. Thus one should consider deleting unused primitives, as well as

replacing or encapsulating them.

In addition to using hand-coding and genetic programming to develop primitives, the combination of techniques lead to cases where primitives were found to be ineffective and thus could be removed. For example, in the filtering membrane problem described above, the primitives `(HasNeighbor direction)` and `(ReadSensorNeighborDist direction)` were not used in early successful solutions. In addition, early attempts to hand-code the solution did not reveal any apparent benefit to them. Thus, they were removed for later runs. A viable solution was found without these primitives.

Consider adding primitives

It may become apparent after some runs that certain primitives might be necessary or helpful. In particular, revisit the points of previous section. In our case, it was at this point that we realized a movement primitive was necessary, even though that could have been clear at the start. Also, some of the gradient primitives could be viewed in this way, since it is arguable that they replace rather than encapsulate the initial message primitives.

5.4 PRIMITIVES FOR RELATED PROBLEMS

Consider using results, or partial results, from similar problems as primitives.

Solutions, or piece of solutions, found by genetic programming to similar problems, including easier versions of problems or related subproblems, can provide useful and robust primitives for more complex problems.

A sorting membrane needs to be able to pass objects through its structure just as a filtering membrane does. The solution found for the filtering membrane problem contains a quick and efficient way to pass objects through a structure of modules. By examining the solution provided by genetic programming, we can remove unnecessary sub-trees to obtain:

```
(If (ReadSensorObjectShouldPass 0.0) (If (If
(ProgN (Move 1.0) (RetractArm 0.8)) (ProgN
(ProgN (Move 1.0) (RetractArm 0.8)) (Move
1.0)) (RetractArm 0.1)) Move 1.0) (Move
0.4)) NormalizeDensity)
```

Although this program does not provide for 100% success on all problems (Kubica and Rieffel, 2002), the result is a relatively efficient filtering program. At the highest level of the primitive is an `If` clause dependent on the `ReadSensorShouldPass` primitive. By simply replacing `(ReadSensorObjectShouldPass 0.0)` with `(ReadSensorIsObject 0.0)` we were able to provide later membrane problems with a robust `(Drop)` primitive.

6 FUTURE WORK

Ideally, a system for automatically generating modular robotic code would only need a description of the capabilities of the modules and the desired behavior for the system to find a solution. Unfortunately, we are currently far from such a system. In particular, current evolutionary approachs are not sufficiently advanced to be able to solve many complex problems on their own.

One difficulty facing such systems is that they are often not capable of deriving an effective set of primitives from the information they are given. Determining fitness functions is also a nontrivial problem, particularly how to reward useful pieces that are not measured by a fitness function coming directly from a problem statement. The problem of modular robot control, particular that of generating effective communication for modular robotic systems, provides a good area in which to explore approaches to this problem.

A first step towards a more automated system is to identify situation in which current systems need help. A better understanding of how a human and a system can collaborate to effectively solve a problem, can lead not only to guidelines that can help humans solve problems more efficiently with the help of machines, but could lead to insights that could ultimately enable the automation of some of the processes currently requiring human input. Our hope is that our effort here not only contributes to this direction, but will also encourage others to do more work along these lines. These problems are difficult, but not so hard that progress cannot be made.

References

P.J. Angeline, J.B. Pollack (1994). *Coevolving High-level Representations.* Artificial Life III, Addison-Wesley, pp. 55-71.

F.H. Bennett III, B. Dolin, E.G. Rieffel (2001). *Programmable Smart Membranes: Using Genetic Programming to Evolve Scalable Distributed Controllers for a Novel Self-Reconfigurable Modular Robotic Application.* Genetic Programming: proceedings of EuroGP 2001, Springer LNCS 2038, pp. 234-245.

F.H. Bennett III, E.G. Rieffel (2000). *Design of Decentralized Controllers for Self-Reconfigurable Modu-*

lar Robots Using Genetic Programming. Proceedings of the 2nd NASA/DoD Workshop on Evolvable Hardware, pp. 43-52.

H. Bojinov, A. Casal, T. Hogg (2000). *Emergent Structures in Modular Self-Reconfigurable Robots.* 2000 IEEE International Conference On Robotics and Automation, pp. 1734-1741.

J.R. Koza (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press

J. Kubica, A. Casal, T. Hogg (2001). *Agent-based Control for Object Manipulation with Modular Self-reconfigurable Robots.* The 2001 International Joint Conference of Artificial Intelligence, pp. 1344-1352

J. Kubica, E.G. Rieffel (2002). *Creating a Smarter Membrane: Automatic Code Generation for Modular Self-Reconfigurable Robots.* To Appear in 2002 IEEE International Conference On Robotics and Automation.

S. Murata, H. Kurokawa, S. Kokaji (1994). *Self-Assembling Machine.* Proceedings of the 1994 IEEE International Conference On Robotics and Automation, pp. 441-448

A. Pamecha, D. Stein, C.-J. Chiang, G.S. Chirikjian (1996). *Design and Implementation of Metamorphic Robots.* Proceedings 1996 ASME Design Engineering Tehcnical Conference and Computers and Engineering Conference, pp. 1 - 10. ASME Press.

S.D. Roberts, D. Howard, J.R. Koza (2001). *Evolving Modules in Genetic Programming Subtree Encapsulation.* Genetic Programming: Proceedings of EuroGP2001, Springer LNCS 2038, pp. 160 - 175.

J.P. Rosca, D.H. Ballard (1996). *Discovery of subroutines in genetic programming.* Advances in Genetic Programming 2. MIT Press.

D. Rus, M. Vona (1999). *Self-Reconfiguration Planning with Compressible Unit Modules.* Proceedings of the 1999 IEEE International Conference On Robotics and Automation, pp. 2513-2520.

D. Rus, M. Vona (2000). *A Physical Implementation of the Self-Reconfiguring Crystaline Robot.* Proceedings of the 2000 IEEE International Conference On Robotics and Automation, vol. 2, pp. 1726 -1733.

W.M. Shen, B. Salemi, P. Will (2000). *Hormones for Self-Reconfigurable Robots.* Proceedings of 6th International Conference on Intelligent Autonomous Systems, pp. 918-925. IOS Press.

W.M. Shen, B. Salemi, Y. Lu, P. Will (2000). *Hormone-Based Control for Self-Reconfigurable Robots.* 2000 International Conference on Autonomous Agents. Barcelona, Spain.

J.W. Suh, S.B. Homans, M. Yim (2002). *Telecubes: Mechanical Design of a Module for Self-Reconfigurable Robotics.* To appear in the 2002 IEEE International Conference On Robotics and Automation.

S. Vassilvitskii, J. Kubica, E. Rieffel, J. Suh, M. Yim (2002). *On the General Reconfiguration Problem for Expanding Cube Style Modular Robots.* To appear in 2002 IEEE International Conference On Robotics and Automation.

D. Wolpert, W. Macready (1995). *No free lunch theorems for search.* Technical Report SFI-TR-05-010.

D. Wolpert, W. Macready (1997). *No free lunch theorems for optimization.* IEEE Transactions on Evolutionary Computation, 1(1) pp. 67 - 82.

M. Yim (1994). *Locomotion with a Unit Modular-Reconfigurable Robot.* Stanford PhD Thesis, 1994.

Convergence Rates for the Distribution of Program Outputs

W. B. Langdon
Computer Science, University College, London, Gower Street, London, WC1E 6BT, UK
W.Langdon@cs.ucl.ac.uk
http://www.cs.ucl.ac.uk/staff/W.Langdon
Tel: +44 (0) 20 7679 4436, Fax: +44 (0) 20 7387 1397

Abstract

Fitness distributions (landscapes) of programs tend to a limit as they get bigger. Markov chain convergence theorems give general upper bounds on the linear program sizes needed for convergence. Tight bounds (exponential in N, $N \log N$ and smaller) are given for five computer models (any, average, cyclic, bit flip and Boolean). Mutation randomizes a genetic algorithm population in $\frac{1}{4}(l+1)(\log(l)+4)$ generations. Results for a genetic programming (GP) like model are confirmed by experiment.

1 INTRODUCTION

We have shown that the fitness distribution of sufficiently large programs will converge eventually [Langdon and Poli, 2002]. We now use standard results from Markov chain theory, to give quantitative bounds on the length of random linear genetic programs, such that the distribution of their outputs is independent of their size. The bounds depend heavily on the type of computer, the fitness function, and scale with the size of the computer's memory. Proving general convergence rates for the fitness of programs requires detailed consideration of the interaction within random programs between different input values. In some cases we can do this, while in others we leave this for future work.

The next section summarises the Markov model of linear genetic programming (GP). Sections 3.1–3.5 describe a wide range of models of computers for running linear GP and prove their convergence properties. Sections 3.1 and 3.2 highlight the importance of internal coupling. Section 3.3 and 3.4 use Markov minorization to prove upper bounds for, firstly any computer (3.3) and secondly average computers (3.4). Section 3.5 is close to some practical GP systems [Banzhaf *et al.*, 1998], while Section 3.6 gives an interesting result on the convergence of bit string genetic algorithms. The application of our results are given in Section 4 and we conclude in Section 5.

2 MARKOV MODELS OF PROGRAM SEARCH SPACES

[Langdon and Poli, 2002] deals with both tree based and linear GP. For simplicity we will consider only large random linear programs. However we anticipate similar bounds also exist for large random trees.

In the following models, the computer is split in two. The random program and all control circuitry form one part, while the computer's *data* memory, inputs and outputs form the second. The memory is treated as a finite state machine (FSM) with 2^N states. (Where N is the number of bits of data in the machine.) Each time a program instruction is executed, data are read from the memory, the result is calculated and written into the memory. This changes the pattern of bits inside the memory. This is modelled as moving the FSM from one state to another. This is deterministic. Given a bit pattern and an instruction, the bit written to memory is also fixed. That is, given a particular state, executing a particular instruction will always move the FSM to the same state. Of course, in general, executing a different instruction will move the FSM to a different state.

Before starting a program, the memory is zeroed and the inputs are loaded[1]. As each instruction in the program is executed, the FSM is updated. If the program is l instructions long, the memory (FSM) is updated l times and then the program *halts*. The program's

[1] Some practical GPs, e.g. Discipulus, write protects the inputs.

answer is then read from the memory. I.e. the output is determined by the last state reached by the FSM.

If there are I instructions then there are I^l possible programs of length l. Suppose we calculate each one's fitness by running it on a fix set of tests and then compare its answers with the tests' target values. The fitness distribution is given by plotting a histogram of the number of programs (divided by I^l) with each fitness value. [Langdon and Poli, 2002] shows, for big enough l, the distribution for one length is pretty much the same as for any another.

It is usually impractical to generate every program of a given length. Instead we consider a large randomly drawn sample of the possible programs. To generate a random program of length l, we simply choose at random l times from the instruction set. When this program is executed, the FSM (i.e. the computer's memory) is updated randomly at each step. But note that the FSM can only move to one of a small number of possible states at each step. Which ones are possible depends *only* on its current state. These are the conditions for a Markov process.

Provided it is possible for a program to set any pattern of bits and there is an instruction which leaves a bit pattern unchanged (no-op), then the Markov process will converge. These conditions mean the FSM is connected, i.e. it is possible to move, in a finite number of steps, from any state to any other. The requirement for at least one no-op keeps the maths simple later by avoiding cycles but its not fundamental. If these conditions hold, then the process of randomly updating the FSM is a Markov process with nice limiting properties. For example, this means if we run the process for long enough (i.e. execute enough random instructions) the probability of the FSM being in any particular state will be a constant. I.e. the probability does not change as more random instructions are executed. (Although it may depend upon which state we are considering.) Secondly it does not depend on how the FSM was started. Since the program's answer is read from the memory, it is determined by the FSM final state. I.e. the probability of any particular answer being given by a random sequence of instructions does not depend on how many instructions there are (provided there are sufficient). As the probability is independent of starting conditions, which include the program's inputs, it does not depend on them either. However if the inputs are write protected, then they are external to the computer's data memory. In which case changes to the inputs have to be considered as changes to the state machine and hence may change the limiting distribution of its outputs.

This convergence applies to the whole of the computer's memory. We expect shorter random programs (i.e. fewer instructions) to be needed if less memory is used. Therefore, depending upon the type of the computer, we might expect much shorter programs to be sufficient to give convergence of just the (small) output register. Indeed, in some cases, we can prove this.

3 CONVERGENCE RESULTS

3.1 SLOW CONVERGENCE EXAMPLE

[Rosenthal, 1995] gives several results on the number of random steps needed by a Markov process to reach equilibrium. In this section we chose what appears to close to a worst case, in order to show an example where the random programs have to be very long indeed. The example is a frog's random walk around a circle of W lily pads. At each time step, the frog can only jump clockwise, anti-clockwise or stay still. [Rosenthal, 1995] uses Markov analysis to show that after sufficient time steps the frog may be found on any lily with equal probability ($\frac{1}{W}$) and to show $O(W^2)$ steps are needed before the chance of any of them being occupied is approximately the same.

We shall use the total variation distance between two probability distributions to indicate how close they are. The total variation distance between probability distributions a and b is defined as $||a - b|| = \sup_{x \subseteq \chi} |a(x) - b(x)|$. I.e. the largest value (supremum) of the absolute difference in the probabilities [Rosenthal, 1995]. The sup is taken over *all subsets*, i.e. every possible grouping of states x, not just single points. (Otherwise it would be small as long as $a(x)$ and $b(x)$ are both always small, even if the distributions a and b are not similar).

Suppose there are three instructions: do nothing, add one to memory and subtract one from memory. We have carry over from one memory word to the next and wrap around if all memory bits are set or all are clear. This corresponds to the frog jumping from lily pad W to 1 or 1 to W. (Remember the lilies are arranged in a circle.) Part of the memory is loaded with inputs and part designated the output register. (Read only inputs are not permitted in this example.)

[Rosenthal, 1995] shows that the actual probability distribution μ_l after l random instructions is exponentially close for large l to the limiting distribution π (in which each of the 2^N states is equally likely). Actually (if there more than two bits of memory, i.e. $N > 2$) we have both lower and upper bounds on the maximum difference (sup) between the actual distribution

of outputs of length l random programs and the uniform 2^{-N} distribution:

$$\frac{1}{2}\left(1-\frac{4\pi^2}{3\ 2^{2N}}l\right) \leq ||\mu_l - \pi|| \leq \sqrt{\frac{e^{-\frac{4\pi^2}{3\ 2^{2N}}l}}{1-e^{-\frac{4\pi^2}{3\ 2^{2N}}l}}}$$

That is the programs have to be longer than $O(2^{2N})$ for the distribution of FSM states to be very close to the limiting distribution. E.g. to make $||\mu_l - \pi|| < 0.1$ the lower bound says l must exceed $0.8\frac{3}{4\pi^2}2^{2N}$, while the upper bound says it need not exceed $\log(101)\frac{3}{4\pi^2}2^{2N}$. For a computer with 1 byte of memory, programs with between 4,000 and 23,000 random instructions need to be considered before each state is equally likely.

The output is read from part of the whole computer's memory (the m bit output register). Since in the limit each of the 2^N states is equally likely, each of the 2^m possible answers is also equally likely. The special instruction set means, the distribution of answers takes just as long to converge as does the whole of the computer. This is despite the fact that the output register only occupies a fraction of the whole of the computer.

The output of any program is $x + p \bmod 2^m$, where x is the input and p is a constant (specific to that program). Note this computer can only implement 2^m functions. The probability distribution of functions clearly follows the distribution of outputs. So when l is long enough to ensure each output is equally likely, then so too is each function.

In general, the distribution of program fitnesses will also take between $0.06\ 2^{2N}$ and $0.35\ 2^{2N}$ to converge (assuming large N). Of course specific fitness functions may converge more rapidly.

3.2 FAST CONVERGENCE EXAMPLE

The second example also uses results from [Rosenthal, 1995] (Bit flipping) [Diaconis, 1988, pages 28–30]. Assume a computer with N bits of memory and $N+1$ instructions. The zeroth instruction does nothing (no-op) while each of the others flips a bit. I.e. executing instruction i, reads bit i, inverts it and then writes the new value back to bit i. Again input (n bits) and output (m bits) registers are defined (and read only inputs are forbidden).

Once again the limiting distribution is that each of the states of the computer is equally likely. However the size of programs needed to get reasonably close to the limit is radically different. Only $\frac{1}{4}(N+1)(\log(N)+c_1)$ program instructions are required to get close to uniform [Diaconis, 1988, page 28] [Rosenthal, 1995]. In fact, for large N, it can also be shown that, in general, convergence will take more than $\frac{1}{4}(N+1)(\log(N)-c_2)$ instructions.

Using the upper bound and setting $c_1 \geq 4$ will ensure we get sufficiently close to convergence. Since then $||\mu_k - \pi|| \leq 10\%$. I.e. random programs of length $\frac{1}{4}(N+1)(\log(N)+4)$ will be enough to ensure each bit of the computer is equally likely to be set as to be clear, regardless of the programs' inputs. (Section 3.5 explains why $c_1 = 4$ is sufficient.) Again in the limiting distribution each state is equally likely.

Returning to our computer with 1 byte of memory, programs with no more than 14 random instructions are needed to ensure each state is equally likely.

Only $m/(N+1)$ bit flips actually effect the output, so $\frac{1}{4}(N+1)(\log(m)+4)$ random instructions will suffice for the each of the 2^m outputs to be equally likely (cf. Section 3.5).

Assume s bits are shared by the input and output registers. We can construct a truth table for each program. It will have 2^s rows. (The non-overlapping bits of the input register are discarded.) The zeroth row gives the output of the program (in the range $0 \ldots 2^m - 1$) when all s bits of the input register are zero. Each bit of the row is equal to the number of times the corresponding memory bit has been swapped by the program, modulo two. Each of $2^s - 1$ other rows is determined by the zeroth row. I.e. the complete table and hence the complete function implemented by a program, is determined by its output with input zero. Therefore 1) for large programs, each of the 2^m functions is equally likely and 2) the distribution of functions converges with the distribution of outputs. Finally the distribution of program fitnesses converges at least as fast. However, since a given fitness function need not treat each of the m output bits equally, its limiting distribution need not be uniform and it can converge faster.

This suggests 9 random instructions will be enough to ensure the output of a 1 byte Boolean (i.e. one bit) computer is random. Further that every Boolean fitness function will also be close to its limiting distribution. Note this does not depend upon the number of input bits n (although n cannot exceed 8 of course).

3.3 ANY COMPUTER

This section gives a quantitative upper bound on the convergence of the distribution of outputs produced by any computer which fits the general framework given in Section 2.

The general Markov minorization condition [Rosen-

thal, 1995] is fairly complex. Fortunately for this proof (and Section 3.4) we can use a simplified special case.

Define P_{ij} to be the probability that starting in state i the next operation will take us to state j. (If j cannot be reached from i in one move, then $P_{ij} = 0$.) The complete matrix P formed from all the P_{ij} is known as the transition matrix. A simple Markov minorization condition is that there is at least one state which can be reached from all the others in one step. That is, there is at least one column of the transition matrix P whose entries are all positive (not zero). Given this the corresponding Markov chain converges geometrically quickly [Rosenthal, 1995].

$$||\mu_k - \pi|| \leq (1 - \beta)^k$$

where

$$\beta = \sum_{y=1..2^N} \min_{x=1..2^N} P(x, y)$$

I.e. β is the sum of the minimum values of the entries in each column of P.

All fine and dandy, however, there are 2^N elements in each column of P but only a small number I of possible instructions. Thus there will be at least $2^N - I$ elements in each column of P that are zero. Thus $\beta = 0$. This does not mean that the Markov process will not converge or even that it will take a long time. It just means the simple application of a minorization condition does not take us very far.

One way round this difficulty is to replace P by P^k in the minorization condition. This means, instead of looking at the available state transitions if each of the I instructions is used once, we consider the transitions possible when they are used k times. For any given state there are up to I^k states the FSM could be in after k instructions. (Ignoring overlaps, each is equally likely.) So if $I^k \geq 2^N$ it is now possible that in at least one column of P there will be no zero entries.

From the way that we constructed our computer, it is possible, eventually, to move from the starting state s_0 to any state y. Let a be the number of steps required. This meets the minorization condition for P^a. In fact $P^a(s_0, y) \geq I^{-a} > 0 \; \forall y$. Therefore $\beta \geq I^{-a}$ and so for any computer:

$$||\mu_k - \pi|| \leq (1 - I^{-a})^{\lfloor k/a \rfloor}$$

Setting $||\cdot||$ to 10% yields a convergence length k for any computer with I instructions $k \leq 2.3025851 a I^a$. Where a is the number of instructions to reach any state. ($a < 2^N$).

3.4 AVERAGE COMPUTER MODEL

Suppose given any possible data in memory each of the I instructions independently randomises it.

Thus for any state x $P(x, y) = 0$ or $1/I$ or $2/I$ or $\ldots$ or I/I. Most elements of the transition matrix $P(x, y)$ will be zero but between 1 and I elements in each column will be non zero. The chance of any given $P(x, y)$ being zero is $(1 - 2^{-N})^I$.

Consider two instructions chosen at random. $P^2(x, y) = 0$, or $1/I^2$ or $\ldots$ or $2I/I^2$. The chance of any given element of $P^2(x, y)$ being zero is $(1 - 2^{-N})^{2I}$.

For l instructions, each element of $P^l(x, y)$ will be a multiple i (possibly zero) of I^{-l}. The values of i will be randomly distributed and follow a binomial distribution with $p = 1/2^N$, $q = 1 - p$ and number of trials $= I^N$. So the distribution of i's mean is $I^l/2^N$ and its standard deviation is $\sqrt{I^l \times 1/2^N \times (1 - 1/2^N)}$. For large I^l the distribution will approximate a Normal distribution. If $I^l \gg 2^N$, even for large 2^N, practically all i will lie within a few (say 5) standard deviations of the mean. I.e. the smallest value of i in any column will be more than $I^l/2^N - 5\sqrt{I^l \times 1/2^N}$. So β will be at least $2^N I^{-l}(I^l/2^N - 5\sqrt{I^l \times 1/2^N})$. I.e. $\beta \geq (1 - 5\sqrt{I^{-l} \times 2^N})$.

Let $\alpha = 5\sqrt{I^{-l} \times 2^N}$. So $\beta \geq (1 - \alpha)$. Next chose a particular value of l so that α is not too small. E.g. set $\alpha = 0.5$ so $\beta \geq 0.5$.

$$\begin{aligned}
\alpha &= 5\sqrt{I^{-l} \times 2^N} \\
\sqrt{I^{-l} \times 2^N} &= \alpha/5 \\
0.5(-l \log I + N \log 2) &= \log(\alpha/5) \\
l &= \frac{-2\log(\alpha/5) + N \log 2}{\log I}
\end{aligned}$$

Now we have a practical value of β we can use the minorization condition on P^l to give

$$\begin{aligned}
||\mu_k - \pi|| &\leq \left(1 - (1 - 5\sqrt{I^{-l} \times 2^N})\right)^{\lfloor k/l \rfloor} \\
&= \left(5\sqrt{I^{-l} \times 2^N}\right)^{\lfloor k/l \rfloor} \\
&= \alpha^{\lfloor k/l \rfloor}
\end{aligned}$$

Choosing a target value of $||\mu_k - \pi||$ of 10% gives:

$$\begin{aligned}
\alpha^{\lfloor k/l \rfloor} &\geq ||\mu_k - \pi|| = 0.1 \\
\lfloor k/l \rfloor \log \alpha &\geq -2.3025851 \\
k &\leq \frac{-2.3025851 \; l}{\log \alpha} \\
&= \frac{-2.3025851 \; (-2\log(\alpha/5) + N \log 2)}{\log \alpha \log I}
\end{aligned}$$

$$= \frac{-2.3025851\ (2\log 10 + N\log 2)}{-\log 2 \log I}$$

$$k \leq \frac{15.298044 + 2.3025851\ N}{\log I} \quad (1)$$

Note this predicts quite rapid convergence for our randomly wired computer. E.g. if it has 8 instructions $k \approx 7 + N$. That is for a one byte 8 random instruction computer programs longer than 16 will be close to the computer's limiting distribution.

Inequality (1) bounds the length of random programs need to be to ensure, starting from any state, the whole computer gets close to its limiting distribution. Again we define parts of the memory as input and output registers. Each program's output is given by m output bits.

Due to the random interconnection of states, on average we can treat each of the 2^m states associated with the output register as projection of 2^{N-m} states in the whole computer, so Inequality (1) becomes $k \leq (15.298044 + 2.3025851\ m)/\log I$. E.g. for Boolean problems ($m = 1$). Only about 9 random instructions are need for an 8 random instruction computer to have effectively reached the programs' outputs limiting distribution.

As in Section 3.2, we can construct a look up table for a particular program which contains the value it yields for each input. It will have 2^n rows, each of which can have one of 2^m values. As in Section 3.2, the relationship between each row is determined by the program. However, the more powerful architecture means that each row can have an apparently independent value. So there are $(2^m)^{2^n}$ possible tables (and hence $2^{m\times 2^n}$ possible functions). For a given input (i.e. row in the lookup table) each output is equally likely. If each row were independent then every complete table (and hence each function) would be equally likely. A loose argument says, we can fill the table by running k random instructions and storing the output register in the table. We then re-use the current contents of the memory (first noting the contents of the input register). We run another k random instructions. This yields another random output value, which is effectively independent of the first. This is stored in the table row corresponding to the intermediate value of the input register. It will take at least 2^n such operations to fill the table but each row will be independent and so each of the $2^{m\times 2^n}$ possible tables will be equally likely. I.e. running $O(2^n m/\log I)$ random instructions will ensure each function is equally likely (cf. no free lunch, NFL [Wolpert and Macready, 1997]). Finally the distribution of program fitness' will also have converged by this point (though its distribution need not be uniform and, for a specific fitness function, it may have converged more quickly).

While such a random connection machine might seem perverse, and we would expect it to be hard for a human to program, on the face of it, it could well be Turing complete (taking into account its finite memory). However since it lacks any particular regularities, we would anticipate random search to be as effective as any other technique (such as genetic programming) at programming it.

3.5 FOUR BOOLEAN INSTRUCTION COMPUTER

This model is the closest to actual (linear) GPs. The CPU has 4 Boolean instructions: AND, NAND, OR and NOR. Before executing any of these, two bits of data are read from the memory. Any bit can be read. The Boolean operation is performed on the two bits and a one bit answer is created. The CPU then writes this anywhere in memory, overwriting what ever was stored in that location before.

Note the instruction set is complete in the sense that, given enough memory, the computer can implement any Boolean function.

As before, we look at the distribution of memory patterns that are produced by running all programs of a given length, l, by considering a large number of random programs of that length. I.e. programs with l randomly chosen instructions.

Each time a random instruction is executed, two memory locations are (independently) randomly chosen. Their data values are read into the CPU. The CPU performs one of the four instructions at random. Finally the new bit is written to a randomly chosen memory location.

Now it considerably simplifies the argument to note that the four instructions are symmetric. In the sense that no matter what the values of the two bits read are, the CPU is as likely to generate a 0 as a 1. That is, each instruction has a 50% chance of inverting exactly one bit (chosen uniformly) from the memory and a 50% chance of doing nothing. Thus we can update the analysis in Section 3.2 based on [Diaconis, 1988, pages 28–30] and [Rosenthal, 1995].

$$||\mu_l - \pi||^2 \leq \frac{1}{4}\sum_{j=1}^{N} \frac{N!}{j!(N-j)!}\left|1 - \frac{j}{N}\right|^{2l} \quad (2)$$

$$= \frac{2}{4}\sum_{j=1}^{\lceil \frac{N+1}{2} \rceil} \frac{N!}{j!(N-j)!}\left(1 - \frac{j}{N}\right)^{2l}$$

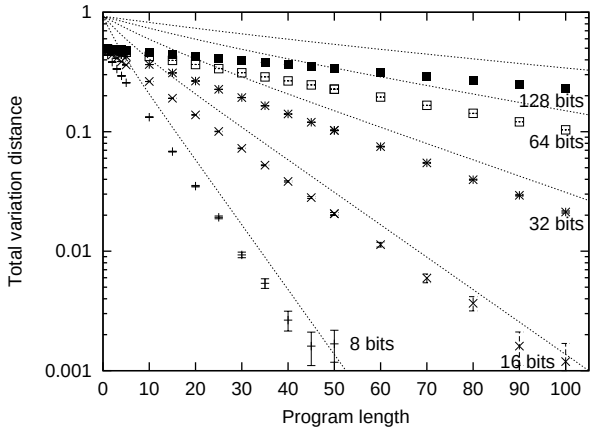


Figure 1: Convergence of outputs of random 3 bit Boolean (AND, NAND, OR, NOR) linear programs with different memory sizes. Note the agreement with upper bound $\sqrt{1/2(\exp(me^{-2l/N})-1)}$ (dotted lines).

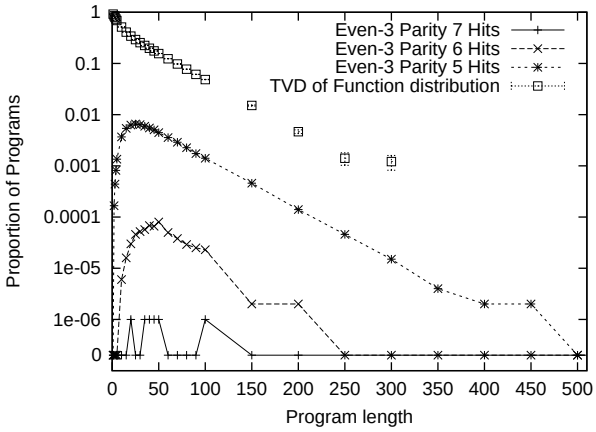


Figure 2: Convergence of Even-3 parity fitness. Even in the limit some functions are more common than others. Longer programs are needed to achieve convergence of functions than of outputs, (Figure 1, 8 bits)

$$< \quad \frac{1}{2}\sum_{j=1}^{\infty}\frac{N^j}{j!}e^{-\frac{2j}{N}l}$$

$$||\mu_l - \pi||^2 \quad \leq \quad \frac{1}{2}\left(e^{Ne^{-\frac{2}{N}}l} - 1\right)$$

Requiring $||\mu_l - \pi||$ not to exceed 10% gives the upper bound $l \leq \frac{1}{2}N(\log(N)+4)$. That is, programs need only be twice as long on this computer (which is capable of real computation) as on the simple bit flipping computer of Section 3.2.

In this computer the chance of updating the output register is directly proportional to its size. So the number of instructions needed to randomised the output register is given by its size (m bits). But we need to take note that most of the activity goes on the other $N-m$ bits of the memory. Therefore Inequality (2) becomes

$$\frac{1}{4}\sum_{j=1}^{m}\frac{m!}{j!(m-j)!}\left|1-\frac{j}{N}\right|^{2l}$$

which leads to $l \leq \frac{1}{2}N(\log(m)+4)$. Figure 1 confirms this.

On this computer, the output of a program given one input is strongly related to its output with another input. This means the loose lookup table argument of Section 3.4 breaks down. The distribution of functions does converge (albeit more slowly than the distribution of outputs) but in the limit each of the $2^{m\times 2^n}$ possible functions are not equally likely (see Figure 2). Detailed modelling of this is left to further work.

How long it takes for a fitness distribution to converge will depend upon the nature of the fitness function. Once its function distribution has converged, the fitness distribution must also converge. However, it could require substantially shorter programs.

3.6 CONVERGENCE IN BIT STRING GAs

The bit flipping model (in Section 3.2) is very close to standard mutation in bit string genetic algorithms (GAs). The principle difference is in GAs the number of bits flipped follows a Poission distribution (unit mean is often recommended [Bäck, 1996]). Thus 0.38 (rather than $1/(l+1)$) of chromosomes are not mutated and 0.26 (rather than zero) chromosomes have two or more bits flipped. (In this section, the length of the bit string chromosome is denoted by l.) Ignoring these differences, it takes only $\frac{1}{4}(l+1)(\log(l)+4)$ mutations to scramble a chromosome from any starting condition.

It is no surprise to find *asymptotic bounds* of $O(l\log(l))$ reported before [Garnier *et al.*, 1999], but note that $\frac{1}{4}(l+1)(\log(l)+4)$ is quantitative and does not require $l \to \infty$. Also it is a reasonably tight bound in the sense that replacing "+4" by a modest negative constant leads to a lower bound. However we include this section mainly because the answer comes straight from standard results without hard work.

Since each chromosome in a GA population is mutated independently, the time taken to scramble an entire GA population is scarcely more than to scramble each of its chromosomes. Crossover makes the analysis more complex but since it moves bit values rather than changing them, we do not expect it to radically change the time needed [Gao, 1998]. E.g. for a GA population

of 32 bit strings, mutation alone (note we turn off selection) will scramble it within about 61 generations. (For standard mutation the value may be slightly different.) Notice this is independent of population size, in contrast the number of generations taken by selection to unscramble the population depends on the size of the population but not l [Blickle, 1996]. According to [Bäck, 1996, Table 5.4] binary tournament selection (without mutation or crossover) takes only 9 generations to remove all diversity from a population of 100.

4 APPLICABILITY

The results in Section 3 refer to specific types of computation, nevertheless we feel they are useful, particularly for common varieties of genetic programming.

The model does not cover programs that contain instructions that are executed more than once. I.e. no loops or backward jumps. (Forward jumps are in principle acceptable, as long as the number of executed instructions remains large.) This is, of course, a big restriction. However, many problems have been solved by GP systems without such loops or recursive function calls [Banzhaf *et al.*, 1998]. The difficulty for the *proofs* is that, in general, repeating (a sequence of) random instructions does not give, on average, the same results as the same number of random instructions chosen independently. (If the loop contains enough random instructions to reach the limiting distribution then the problem does not arise because the input to the next iteration to the loop is already in the limiting distribution and so will remain there.) Similarly, there is no problem if the loop is followed by a large number of random instructions.

While the proofs suggests that the program will halt after l instructions, they can be made slightly more general by extracting the answer from the output register after l time intervals, allowing the program to continue (or to be aborted). These have been called "any time algorithms". They have been used in GP, e.g. [Teller, 1994].

The dominant factors in determining length required for near convergence are the type of computer considered and the size of its (data) memory. The scaling law is given by the type. Comparing the four types in Section 3 suggests that the degree of interconnections in the state space is the important factor. The ability to move directly from one memory pattern to another leads to linear scaling, while only being able to move to 2 adjacent data patterns lead to exponential scaling. We suggest that the "bit flipping" and "4 Boolean Function" models are more typical and so we suggest $O(N \log N)$ would be found on real computers.

Most computers support random access at the byte or word level. This would suggest N should be the number of bytes or words in the data memory. However then we would expect the individual bits in each byte or word to be highly correlated, and so we would anticipate the simple $O(N \log N)$ law would break down. I.e. further random instructions will be required to randomise them. This might result in a multiplicative factor of $8 \log 8$ or $32 \log 32$ but this yields the same scaling law ($(8 \log 8)N/8 \log N/8 = \mathrm{O}(N \log N)$) possibly with different numerical values.

Some linear GP systems write protect their inputs. The proofs can be extended to cover this by viewing the read-only register as part of the CPU (i.e. not part of the data memory). Then we get a limiting distribution as before, but it depends on the contents of the read-only register, i.e. the programs' input. In general we would expect this to give the machine a very strong bias (i.e. an asymmetric limiting distribution) and in some cases this might be very useful.

All of the calculations in Section 3 have been explicitly concerned with the distribution of answers produced by the programs and the functions implemented by them. In principle we can use the Markov arguments to consider the distribution of functions implemented by the programs in other types of computer. The Markov process now becomes a sequence of changes in function. We start with the identity function and the distribution of functions rapidly spreads through the $2^{N^{2^N}}$ functions. An obvious difficulty is that the size of the transition matrixes increases exponentially (from $2^N \times 2^N$ to $2^{N^{2^N}} \times 2^{N^{2^N}}$). This might lead to an exponential (or worse) increase the upper bound scaling laws.

The random computer (cf. Section 3.4) gives an interesting model. Indeed it represents the average behaviour over all possible computers (of this type).

Finally an alternative view is to treat random instructions as introducing noise. Some instructions, e.g. clear, introduce a lot of noise, while others e.g. NAND, introduce less. So we start with a very strong, noise free, signal (the inputs) but each random instruction degrades it. Eventually, in the limiting distribution, there is no information about the inputs left. Thus the entropy has monotonically increased from zero to a maximum.

5 CONCLUSIONS

The distribution of outputs produced by all computers converges to a limiting distribution as their (linear) programs get longer. We provide a general quantitative upper bound ($2.31aI^a$, where I is the number of instructions and a is the length programs needed to store every possible value in the computer's memory, Section 3.3). Tighter bounds are given for four types of computer. There are radical differences in their convergence rates. The length of programs needed for convergence depends heavily on the type of computer, the size of its (data) memory N and its instruction set.

The cyclic computer (Section 3.1) converges most slowly, $\leq 0.35\ 2^{2N}$, for large N. In contrast the bit flip computer (Section 3.2) takes only $\frac{1}{4}(N+1)(\log(m)+4)$ random instructions (m bits in output register). However in both, the distributions of outputs and of functions converge at this same rate to a uniform limiting distribution.

In Section 3.4 we introduced a random, model of computers. This represents the average behaviour over all computers (cf. NFL [Wolpert and Macready, 1997]). It takes less than $(15.3 + 2.3m)/\log I$ random instructions to get close to the uniform output limit. However a less formal arguments suggests a multiplicative factor of 2^n needs to be included before the distribution of functions is also close its limit.

Section 3.5 shows the output of programs comprised of four common Boolean operators converges to a uniform distribution within $\frac{1}{2}N(\log(m) + 4)$ random instructions. The importance of the pragmatic heuristic of write protecting the input register, is highlighted, since without it there are no "interesting" functions in the limit of large programs.

Section 3.6 shows the number of generations $(\frac{1}{4}(l + 1)(\log(l) + 4))$ needed for mutation alone to randomise a bit string GA (chromosome of l bits).

Practical GP fitness functions will converge faster than the distribution of all functions, since they typically test only a small part of the whole function. Real GP systems allow rapid movement about the computer's state space and so appear to be close to the bit flipping (Section 3.2) and four Boolean instruction (Section 3.5) models. We speculate rapid $O(|\text{test set}|N \log m)$ convergence in fitness distributions may be observed.

It is ten years since Jaws 1, these are the first general quantitative scaling laws on the space that genetic programming searches. They provide theoretical support for some pragmatic choices made in GP.

Acknowledgments

I would like to thank Jeffrey Rosenthal, David Corney, Tom Westerdale, James A. Foster, Riccardo Poli, Ingo Wegener, Nic McPhee, Michael Vose and Jon Rowe.

References

[Bäck, 1996] Thomas Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press, New York, 1996.

[Banzhaf *et al.*, 1998] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, 1998.

[Blickle, 1996] Tobias Blickle. *Theory of Evolutionary Algorithms and Application to System Synthesis*. PhD thesis, Swiss Federal Institute of Technology, Zurich, November 1996.

[Diaconis, 1988] Persi Diaconis. *Group Representations in Probability and Statistics*, volume 11 of *Lecture notes-Monograph Series*. Institute of Mathematical Sciences, Hayward, California, 1988.

[Gao, 1998] Yong Gao. An upper bound on the convergence rates of canonical genetic algorithms. *Complexity International*, 5, 1998.

[Garnier *et al.*, 1999] Josselin Garnier, Leila Kallel, and Marc Schoenauer. Rigorous hitting times for binary mutations. *Evolutionary Computation*, 7(2):173–203, 1999.

[Langdon and Poli, 2002] W. B. Langdon and Riccardo Poli. *Foundations of Genetic Programming*. Springer-Verlag, 2002.

[Rosenthal, 1995] Jeffrey S. Rosenthal. Convergence rates for Markov chains. *SIAM Review*, 37(3):387–405, 1995.

[Teller, 1994] Astro Teller. Genetic programming, indexed memory, the halting problem, and other curiosities. In *Proceedings of the 7th annual Florida Artificial Intelligence Research Symposium*, pages 270–274, Pensacola, Florida, USA, May 1994. IEEE Press.

[Wolpert and Macready, 1997] David H. Wolpert and William G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, April 1997.

Is the Perfect the Enemy of the Good?

Sean Luke
George Mason University
http://www.cs.gmu.edu/∼sean/

Liviu Panait
George Mason University
http://www.cs.gmu.edu/∼lpanait/

Abstract

Much of the genetic programming literature compares techniques using counts of ideal solutions found. These counts in turn form common comparison measures such as Koza's Computational Effort or Cumulative Probability of Success. The use of these measures continues despite past warnings that they are not statistically valid. In this paper we too criticize the measures for serious statistical problems, and also argue that their motivational justification is faulty. We then present evidence suggesting that ideal-solution counts are not necessarily positively related to best-fitness-of-run statistics: in fact they are often inversely correlated. Thus claims based on ideal-solution counts can mislead readers into thinking techniques will provide superior final results, when in fact the opposite is true.

1 INTRODUCTION

> The best is the enemy of the good.
> — Voltaire (1694–1778)

> He who is determined not to be satisfied with anything short of perfection will never do anything to please himself or others.
> — William Hazlitt (1778–1830)

The research methodology in the genetic programming (GP) has many unusual features. Some of these features are good. Some are not. But we tend to stick with the bad ones out of inertia: we do it that way because others did. Surprisingly, the literature does not have a large number of critics of the existing methodology. One notable exception is Jason Daida, who has criticized poor random number generator usage [1997], evaluation and verification methodology [1999a], and historical metaphors [1999b]. Paterson and Livesey [2000] have decried the poor statistics behind many claims, noting that many papers do no means testing at all. Angeline [1996] has criticized the statistical reliability of Koza's Cumulative Probability of Success measure, a criticism echoed in [Paterson and Livesey 2000].

Here we will continue the criticism of the popular Cumulative Probability of Success and other measures based on counting the number of ideal solutions discovered. There are serious statistical flaws with such measures, but that is not all. These measures also have questionable motivational philosophy, and most importantly, they are poorly correlated with other more accepted measures of run quality in the evolutionary computation community.

This paper was born out of experiments for another purpose: to test whether fitness might be improved and tree size reduced by increasing the noise of a GP breeding operator. The operator chosen was subtree crossover, and it was made noisier through increasing the number of times two parents were crossed over to create a child. Crossing over more times does in fact decrease the mean tree size by statistically significant amounts, but it also worsens the best fitness of the run by a statistically significant margin. But these experiments yielded another odd fact: ideal solution counts were not necessarily tied to fitness results, and in some cases were inversely correlated with them.

The remainder of the paper will discuss common ideal-solution count measures and their statistical weaknesses, and question the motivational philosophy behind them. Then the paper will present the evidence stemming from these experiments. The paper then finishes with discussion and recommendations.

2 A TALE OF TWO MEASURES

The non-GP evolutionary computation literature has traditionally compared techniques using the mean best fitnesses of a large ($>$30) sample of runs per technique, accompanied with so-called "best-so-far-curves" (plots of the

mean best fitness discovered so far), plus a t-test[1] or other difference-of-means test. When generalizability is important, the mean best fitness results are supplemented with generalization ratings on a test set.

Koza [1992] presented a very different metric for computing the "quality" of an evolutionary procedure: how often and how rapidly it discovered the ideal solution. From this data were derived a variety of statistical metrics ultimately computing how many individuals would need to be evaluated before an ideal solution was expected to be found with some probability.

Koza defined four dependent statistical measures, given as follows. The *instantaneous probability of success* measure $Y(m,i)$ is the probability that a run with a population size of m will discover an ideal solution for the first time on generation i. From this it is simple to determine $P(m,i)$, the *cumulative probability of success*, which is the probability that a run will discover an ideal solution before or on generation i. Koza writes $R(m,i,z)$ as the number of evolutionary runs required to have a probability $z = 99\%$ of discovering a solution before generation i. He defines this as

$$R(m,i,z) = \left\lceil \frac{\log(1-z)}{\log(1-P(m,i))} \right\rceil$$

Sometimes this is shortened to just $R(z)$. The *individuals to be processed* measure $I(m,i,z)$ is then defined straightforwardly as $I(m,i,z) = m(i+1)R(m,i,z)$, at least for generational evolutionary procedures. The *computational effort* measure E is the minimum of $I(m,i,z)$ over all values of i.

Ideal-solution count measures have since taken root in the GP community. Of these four, the two most common measures in the literature are Cumulative Probability of Success, for which higher values are better, and its derived measure Computational Effort, for which lower values are better. We performed an informal survey of the genetic programming and evolvable hardware non-poster papers in the three GECCO conferences so far (1999, 2000, 2001). Of these, 44 compared two or more techniques for solution quality. Eighteen compared the mean best fitness of run between techniques.[2] Twelve instead used statistics based on the number of ideal solutions discovered (most used the Koza measures). Fourteen used a train/test methodology borrowed from the machine learning community: typically they measured how long it took to discover a perfect solution to a training set, then tested generalization ability.[3]

We feel there are three problems with ideal-solution count measures. First, they rely on unacceptably poor statistics, at least as generally practiced in the community, and to remedy this would require a very large sample size. Second, they are founded on, in our opinion, an unclear motivational philosophy. The third problem, and most troubling, is that they appear to be uncorrelated with best-fitness-of-run comparisons. These problems call a fair chunk of the literature into question.

2.1 STATISTICAL PROBLEMS

Ideal solution count measures are statistically suspect.

First, the measures are based on a single point sample of the number of ideal solutions and the generations in which they were found. A point sample does not have a mean test: there is no accepted procedure to state that two such samples differ in a statistically significant way.

The point sample difficulty might be alleviated by doing a large number of runs, then dividing them into at least thirty piles, then counting the ideal solutions in each pile and using the mean number of ideal solutions per pile as a sample statistic. For Symbolic Regression this is feasible, as a large number of runs end in perfect solutions. But for many problem domains, ideal solutions are typically few and far between. In this paper we gathered samples of five hundred each, typically five to ten times the number found in most of the experimental literature. And still, the ideal-solution counts for the Artificial Ant and 11-Bit Boolean Multiplexer were so small (at most sixteen and twelve respectively) that dividing into piles was not doable.

Second, several of the measures are statistically dependent across generations. For example, to truly compute statistically independent Cumulative Probability of Success measures for both generation 4 and for generation 8 would require two separate, independent samples.

Third, changes in the Individuals to be Processed measure and its derived Computational Effort measure are both greatly exaggerated when small changes occur in ideal so-

[1] The validity of the t-test and ANOVA for GP were examined in [Paterson and Livesey 2000]. As the t-test and ANOVA assume normality, the authors had expected to find them wanting in the GP realm, but astonishingly they significantly outperformed non-parametric tests. Even so, the authors warned about the dangers of relying too much on t-tests for the skewed distributions common to GP. We agree! Still, we think that t-tests and ANOVAs should at least be the *bare minimum* for means testing in GP.

[2] Of the eighteen experiments which used fitness curves to compare techniques, only seven used good statistics. The other eleven had sample sizes that were too small (much less than 30) or unreported, or they did not indicate statistical significance results or variance information. One of us (Sean Luke) hastens to note that he too has published papers with statistical difficulties, though in his defense they were either corrected ([Luke and Spector 1997] fixed in [Luke and Spector 1998]) or openly acknowledged and justified in the paper itself [Luke 2001b].

[3] As many machine learning algorithms are deterministic, they do not require statistical mean tests. This is not true for stochastic evolutionary computation algorithms: yet relatively few train/test methodology papers in this survey presented statistical significance results.

lution counts. For example, note that the Computational Effort in Figure 12 is a strongly nonlinear function of the number of ideal solutions found, shown in Figure 10.

To overcome all the statistical problems detailed here would require multiple independent samples across generations, plus dramatically larger sample sizes for the more difficult domains. We feel these statistical flaws would be forgivable only if ideal-solution counts were the *only feasible* way to compare techniques. But they are not.

2.2 MOTIVATIONAL PHILOSOPHY

The evaluation functions used in the assessment of a GP individual's fitness do not correspond well to a stated goal of an ideal-solution end result. GP optimizes for as good a fitness as it can get, not for increasing probability of attaining the ideal. Many GP problem domains are highly deceptive, leading the evolutionary trajectory away from the ideal rather than toward it. Consider the Symbolic Regression domain: if the ideal solution is not found early on, and the population has fixated on certain near-solutions, it will continue to tack on code to the bottom of trees which can make the solutions only incrementally fitter. Many Symbolic Regression runs will ultimately generate very large trees with solutions very close to the answer, but far (in makeup) from anything remotely resembling *the* answer.

Given this, and given the statistical problems behind ideal-solution count measures, what is the GP community's motivational justification for using ideal-solution counts at all? We believe that counts are popular because of a philosophical conceit that GP operates over problem domains which demand *correct programs*. More generally this can be thought of as an absolute hard constraint on the desired outcome: either the program works or it doesn't work, and a highly fit but suboptimal solution is not valuable. A term peculiar to GP, "discovery", reinforces this notion: either GP "discovers" the answer, or it doesn't.[4]

[4]In fact, though his influential text introduced the ideal-solution count measures discussed in this paper, Koza couched his support of this philosophy, writing:

"Several pages ago, when I spoke of writing a computer program to center the cart in optimal time, you probably assumed that I was talking about writing a *correct* computer program to solve the problem. Nothing could be further from the truth. In fact, this book focuses almost entirely on *incorrect* programs. In particular, I want to develop the notion that there are gradations in performance among computer programs. Some incorrect programs are very poor; some are better than others; some are approximately correct; occasionally, one may be 100% correct. Expressing this biologically, one could say that some computer programs are fitter than others in their environment. It is rare for any biological organism to be optimal." [Koza 1992, p. 130]

Regression	3	2	4	1	5	6	7	10	8	9
Multiplexer	2	1	3	4	5	6	7	8	9	10
Ant	2	1	4	6	3	5	7	8	9	10

Table 1: Statistical significance groupings for the mean best-fitness-of-run for each problem domain, using the Tukey post-hoc ANOVA test. Numbers indicate the number of crossovers for a given multi-crossover technique. Techniques are ordered by increasing (poorer) mean best-fitness-of-run. Bars connect techniques with statistically insignificant differences in means among them. Note that more crossovers generally results in worse mean best-fitness-of-run. Compare to Figures 1, 5, and 9.

It is clear that there exist valid and important GP problem domains where discovery is of paramount importance. However, we believe that many, and likely most, new problems in GP rarely require 100% correctness as a necessary attribute. These problems include neural networks, molecular structures, soccer softbot programs, probabilistic and quantum algorithms, analog electrical circuits, etc. The primary reason for this, we think, is that these new domains are significantly harder and their optima are often unknown. Discovering the optimum, and particularly discovering it enough times to make statistical conclusions, is a luxury usually reserved for only the simplest of problem domains.

We are now out of the proof-of-concept period for GP. For the technique to now be used realistically as a *tool*, we must assume it will typically be used to attack hard problems for which we do not know the optimum, do not expect it to discover the optimum, nor even know if there *is* an optimum. An engineer would ask: if we already know the answer, why bother to use GP to find it? What matters is not if technique A finds more perfect solutions than technique B does to Easy Problem C. What usually matters is that technique A gets a better answer than B does for Hard Problem D. We submit that if one can "discover" the optimum enough times to validly measure the performance of a technique against a given problem domain, then we are dealing with a toy problem.

If past literature used a weak methodology, we should discontinue its use. Nonetheless, we recognize that for those problems which demand perfection, ideal-solution-count statistics may have some usefulness. Later in this paper we will recommend experimental protocols which may incorporate ideal solution counts as one part of a comprehensive analysis. At the same time, we will propose an alternative which we think provides more useful information.

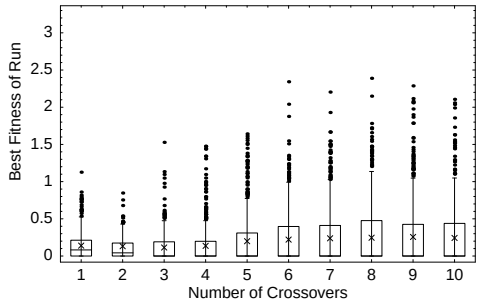


Figure 1: Boxplots of the distribution of best-fitness-of-run, by number of crossovers, Symbolic Regression domain. Lower fitness is better. Compare to Table 1.

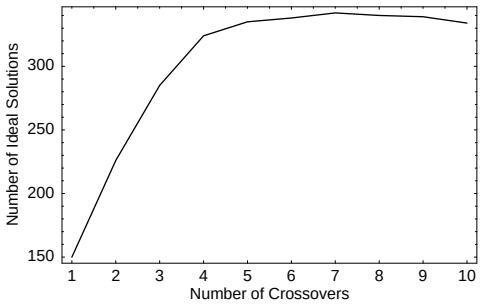


Figure 2: Number of ideal solutions found, by number of crossovers, Symbolic Regression domain.

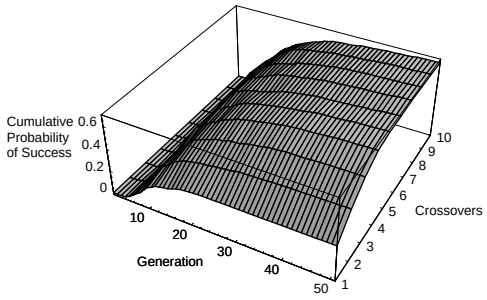


Figure 3: Cumulative Probability of Success per generation, by number of crossovers, Symbolic Regression domain.

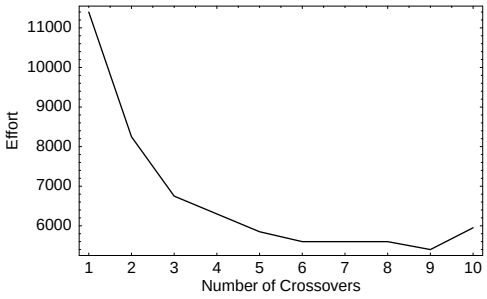


Figure 4: Computational Effort by number of crossovers, Symbolic Regression domain.

3 A TROUBLING LACK OF CORRELATION

At first it seems intuitive that a system which searches better for high-fitness solutions would also tend to find more ideal solutions. An experiment gone awry shows us that in fact this is not necessarily the case.

The experiment we performed applied what we call *multi-crossover* to GP. Ordinarily GP crossover selects two individuals, and performs one swap of randomly-chosen subtrees, producing two children. If a child passes validity constraints (such as maximal depth), it then enters the next generation; otherwise a copy of the child's mother enters in its stead. Multi-crossover is simply a composition of GP crossover operators: two parents are selected and crossover is performed, including validity constraints. Then the children become the "parents" for the next crossover operator, which produces two new children. This happens N times, then the final results enter the next generation.

We had two reasons for doing multi-crossover experiments. First, some models of code bloat (our own depth-based theory [Luke 2000], Defense Against Crossover [Banzhaf et al. 1998], and Removal Bias [Langdon et al. 1999]) assume that there is a single crossover per individual; hence the probabilities of choosing crossover point A or B respectively were not independent. While increasing the number of crossover events would not break this dependency, it could lower the effect the dependency had in causing bloat.

Second, adding more nonhomologous crossover events meant adding more variation (more noise) into the breeding procedure. This gave us a dial to turn which would effectively change the amount that crossover "randomized" individuals (an idea inspired by arguments made in [Angeline 1997]). Would more randomization be beneficial or detrimental to GP?

The experiments were done as follows. We ran for 51 generations, including the initial generation, using a population of 500, and tournament selection of size 7. Multi-crossover was the sole operator used. The three problem domains chosen were Symbolic Regression, 11-Bit Boolean Multiplexer, and Artificial Ant. Symbolic Regression used no Ephemeral Random Constants. Artificial Ant used the Santa Fe Trail. All other run and problem domain parameters were done as stipulated in [Koza 1992]. The evolutionary computation system used was ECJ [Luke 2001a].

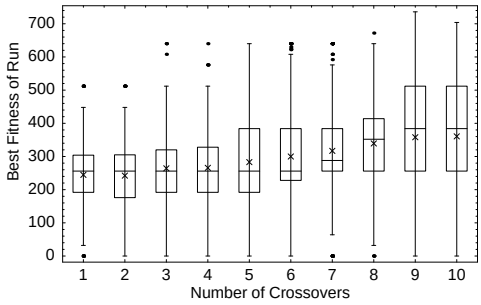


Figure 5: Boxplots of the distribution of best-fitness-of-run, by number of crossovers, 11-Bit Boolean Multiplexer domain. Lower fitness is better. Compare to Table 1.

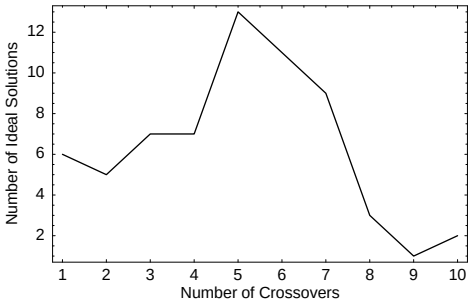


Figure 6: Number of ideal solutions found, by number of crossovers, 11-Bit Boolean Multiplexer domain.

We performed ten different experiments, with multi-crossover set to 1 through 10 crossovers respectively. Each experiment consisted of 500 independent runs. This number of runs is much higher than is necessary to produce best-fitness-of-run results, but we needed as many runs as possible to feel at least partially confident in our ideal-solution counts, and 15,000 total runs was the most we could afford to do. For each problem domain we used boxplots[5] to plot best-fitness-of-run distributions for different numbers of crossovers. We also plotted the number of ideal solutions found, and the Cumulative Probability of Success and Computational Effort measures.

3.1 SYMBOLIC REGRESSION RESULTS

Symbolic Regression did in fact lower tree size. But it did so at the cost of a statistically significant worsening of fitness, though only gradually: large swaths of crossover experiments were in the same statistical equivalence class, as shown in Table 1. Increasing the amount of noise in the crossover procedure, thus moving the system more towards

[5] In a boxplot, the rectangular region covers all values between the first and third quartiles, the stems mark the furthest individual within 1.5 of the quartile ranges, and the center horizontal line indicates the median. Dots show outliers, and $\times$ marks the mean.

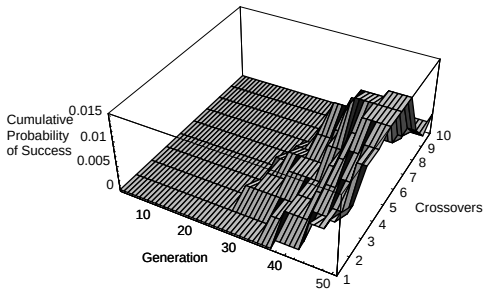


Figure 7: Cumulative Probability of Success per generation, by number of crossovers, 11-Bit Boolean Multiplexer domain.

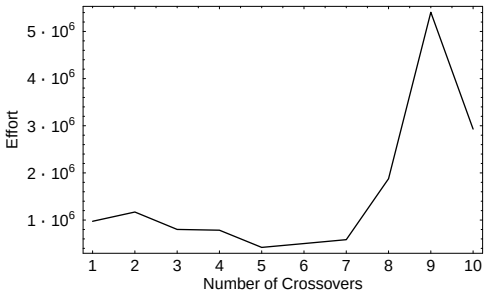


Figure 8: Computational Effort by number of crossovers, 11-Bit Boolean Multiplexer domain.

random search, yielded worse results on average, using the mean best-fitness-of-run measure as shown in Figure 1.

But one would not have known this from the ideal-solution counts. As noise in crossover increased, the number of ideal solutions increased rapidly from 150 per 500 runs with a single crossover, to stabilizing at about 350 per 500 runs with seven crossovers or more, as shown in Figure 2. This in turn resulted in an unexpected Cumulative Probability of Success curve, and a major decrease in Computational Effort as the number of crossovers increased, as shown in Figures 3 and 4.

Symbolic Regression is the easiest of the three problem domains presented: it finds the ideal solution very often (no less than 30% of the time in these experiments). Thus our counts were very high and the Cumulative Probability of Success curve was very smooth.

This outcome was very disturbing. Were we to have used ideal-solution count measurements as our basis of comparison in this experiment, our conclusions wouldn't just have been uncorrelated with best-fitness-of-run results: they

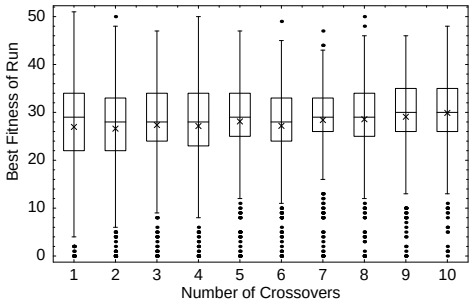


Figure 9: Boxplots of the distribution of best-fitness-of-run, by number of crossovers, Artificial Ant domain. Lower fitness is better. Compare to Table 1.

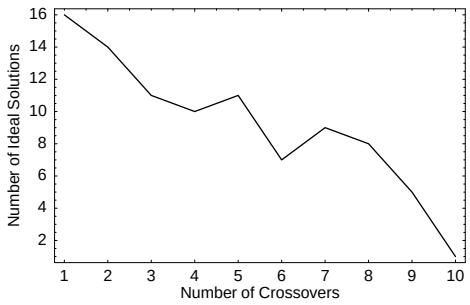


Figure 10: Number of ideal solutions found, by number of crossovers, Artificial Ant domain.

would have been the *opposite*. We would have concluded that increasing number of crossovers does a better job.

3.2 11-BIT BOOLEAN MULTIPLEXER RESULTS

11-Bit Boolean Multiplexer was to yield another surprise. First, it too lowered tree size, and like Symbolic Regression, it did so by statistically significantly worsening the fitness results. Again, increasing the amount of noise yielded worse results on average, when using the mean best-fitness-of-run measure, as shown in Figure 5. This time, the fitnesses worsened rapidly, with few in the same statistical equivalence class, as seen in Table 1.

Given its tendency to bloat like Regression does, we fully expected Multiplexer to have similar ideal-solution counts. But this was not quite the case. As the number of crossovers increased, the number of ideal solutions increased, but then it then decreased again. Multiplexer is a relatively more difficult problem domain to find ideal solutions in: thus we found no more than thirteen ideal solutions in 500 runs, with the maximum peaking when we applied five crossovers. The nadir was a single solution discovered, when we applied nine crossovers, as shown in Figure 6.

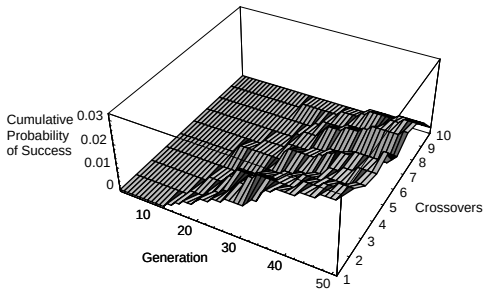


Figure 11: Cumulative Probability of Success per generation, by number of crossovers, Artificial Ant domain.

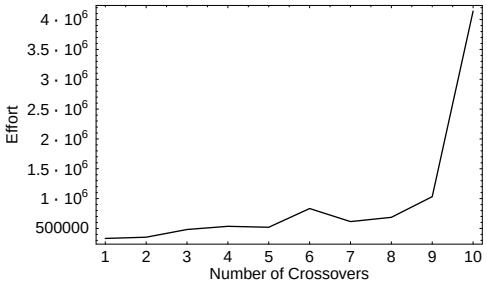


Figure 12: Computational Effort by number of crossovers, Artificial Ant domain.

The Cumulative Probability of Success similarly followed a bell curve distribution, as is shown in Figure 7. The relatively low number of ideal solutions resulted in the expected exaggerated swings in Computational Effort, as shown in Figure 8.

Here, while a best-fitness-of-run measure would state that increasing crossovers generally decreased quality, an ideal-solution count measure would argue that increasing crossovers somewhat (to five) made the result twice as good.

3.3 ARTIFICIAL ANT RESULTS

Artificial Ant gave us yet *another* result. Again, it lowered tree size, and once again, it also did so by statistically significantly worsening the fitness results (Figure 9) though like Symbolic Regression the results worsened only slightly, resulting in large numbers of experiments in the same statistical equivalence class (Table 1).

Figure 10 reveals that this time, the ideal-solution counts followed the statistically significant fitness: increasing the amount of noise decreased the number of ideal solutions

discovered. As shown in Figure 11, the Cumulative Probability of Success followed a radically different curve than was the case in Symbolic Regression. Like Multiplexer, Artificial Ant discovers relatively few ideal solutions (no more than sixteen out of 500). And like Multiplexer, this exaggerates the Computational Effort results (Figure 12).

Here, as in Symbolic Regression, one would conclude on the basis of the best-fitness-of-run measure that the mean is getting just a little worse, though statistically significantly. But here an ideal-solution-count measure suggests that increasing the number of crossovers considerably worsens the result.

4 WHAT'S GOING ON?

For all three problems, fitness gradually worsened as the number of crossovers increased. But each problem yielded a wildly different result in the ideal-solutions count, and thus in the Cumulative Probability of Success and Computational Effort metrics. In short, the ideal-solution counts were not correlated with the best-fitness-of-run measure.

We think the reason for this phenomenon is that, as shown in Figures 1, 5, and 9, the number of ideal solutions is more closely linked to the variance of — rather than the mean of — the best-fitness-of-run distribution. As the variance increases, fitness is increasingly scattered both up and down. But there is a bound at zero (one cannot be better than perfect), whereas there is no bound on getting worse. Thus a wider variance tends to rack up more perfect scores. This trend is echoed in the data for all three problem domains, although the ideal-solution counts in the Artificial Ant and Multiplexer domains are low enough to make us wary about making a pronouncement.

In the Symbolic Regression domain, the distributions are skewed. As the number of crossovers increases, the mean increases a little, but not nearly as much as the variance does. Thus a worsening in the mean is not able to prevent ideal solutions from piling up. In the Multiplexer domain, the mean worsens only a little initially but rapidly at the end, while the variance generally increases steadily. This allows the mean to lose to, then catch up with the variance, which might explain the ideal-solution counts. Lastly, the Artificial Ant domain's mean worsens slowly while the variance *decreases*, which can explain the steady decrease in ideal-solution counts.

This combination of mean and variance has a closely related effect: even if low-variance technique A has a higher expected value than high-variance technique B for a single run, B can still have a higher expected best result of N independent runs (Figure 13). This phenomenon was explored in [Luke 2001b].

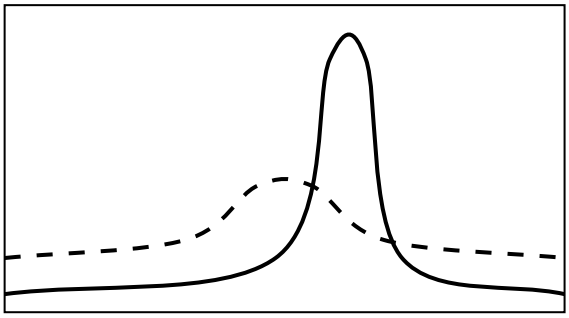


Figure 13: Example of two distributions which differ in mean and in variance. While one sample from the flatter distribution will give a lower expected result (the mean) than the tall distribution, in this case the best of two or more samples from the flatter distribution will give a *higher* expected result.

We will not venture a guess as to *why* increasing the number of crossovers changed the mean and variance results in the way it did. We had expected that increasing the number of crossovers would consistently increase the variance, but clearly it does not.

Regardless of the outcome, we do not believe that these experiments bode well for the body of GP literature which relies on ideal-solution counts to compare the quality of different techniques. We are concerned that upon reading this literature, experimenters may mistakenly conclude that certain techniques are better than others on average, when in some cases better ideal-solution counts are actually indicative of a *worse* result in mean best-fitness-of-run.

5 RECOMMENDATIONS

One downside to mean-best-fitness-of-run results is demonstrated in the Symbolic Regression domain: techniques which produce large, bloated trees that are functionally "close" to the ideal are given high marks, even though the results do not remotely resemble our notion of what the ideal individual ought to look like. Ideal-solution counts can be somewhat useful here in weeding these pretenders out. But given their statistical problems, we feel that ideal-solution count results alone simply cannot be justified.

If such incrementalism is the critical sticking-point tempting an author to only report ideal-solution counts, we suggest instead that the author produce results showing both mean best fitness of run *and* mean tree size. If a technique is superior both in tree size and mean best fitness, it is more difficult to argue that the improvements in mean best-fitness-of-run are due to incrementalism, since this would likely also increase tree size. Another approach would be

to change the methodology to one showing generalization, using training and testing sets. Incrementalism tends to result in solutions custom-tailored to the training set: such poor generalizability would then show up when the final solution is compared to the testing set.

If instead one were performing runs in a domain which demanded perfection, then it would obviously be useful to know how often a technique was likely to find the ideal solution. Another situation also commonly arises: trying to beat an existing record, where it's useful to know not if a technique is likely to find the ideal, but whether or not it is likely to find any solution better than the record. In these situations we suggest that counts could supplement, but not replace, the mean best-fitness-of-run results.

But for most cases we suggest a different point statistic to use as a supplement. The procedure specified in [Luke 2001b] gives the expected maximum best-fitness-of-run for N total runs. This procedure, like the ideal-solution-count procedure, suffers statistically from the assumption that the sample is exactly representative of the population, and from its reuse of statistically dependent data. However, as a supplement to mean best-fitness-of-run results, we think it is a more useful metric than ideal solution counts in most cases.

The procedure is as follows. Perform a large number m runs for a given technique T. Sort the runs by fitness and assign ranks $1,...,m$ to the runs, where rank 1 is the worst-fitness run for the technique. Let $F(r)$ be the fitness of the run ranked r. Then if one were to perform N runs and return the best run, the expected best fitness $E(T,N)$ among those N runs would be:

$$E(T,N) = \sum_{r=1}^{m} F(r) \frac{r^N - (r-1)^N}{m^N}$$

This presumes that higher fitness values are better. Now consider two techniques A and B, where beyond some point $N \geq C$, $E(B,N)$ is consistently greater than or equal to $E(A,N)$. Ideally, C would be 1. This would lend evidence to the belief that not only is A better than B on average, but it is also superior no matter how many runs you are likely to perform. This gives strong weight to the claim that A really is better than B. Further, it seems likely that if technique A finds many more ideal solutions than technique B, that $E(A,N)$ will surpass $E(B,N)$ above some point N where finding ideal solutions with A becomes sufficiently common.

In any case, given their grievous statistical problems, we strongly urge that an ideal solution count measures never be used alone as proof that one technique is superior to another, except under special circumstances and with the appropriate disclaimers. If used at all, they should be only used to bolster a more statistically viable fitness comparison procedure.

Last, we recommend that experimenters more closely adopt difference-of-means tests (at least t-tests and ANOVAs), and a reasonable sample size (30 at a minimum) in published evolutionary computation experiments, or give justifications for doing otherwise.

6 CONCLUSION

Many papers in the GP literature use ideal-solution counts in one way or another, usually to compare the quality of techniques. This notion was popularized by Koza's Cumulative Probability of Success, Individuals to be Processed, and Computational Effort measures. This use continues despite warnings that counts are poor estimates, as they are a point statistic with no associated means test; as they make assumptions about dependencies across generations; and as they are exaggerated by small count sizes.

In this paper we reiterated this warning about counts, and noted motivational concerns. Specifically, we noted that for most "difficult" problems, the goal is to find *as good* a solution as possible. Counting the number of times the ideal solution is found does not help achieve this goal. Further, if one can find the ideal reliably, then the problem is trivial.

We also demonstrated the disturbing fact that ideal solution counts are not well correlated with mean best-of-fitness measures. In fact, for some problem domains, we showed that ideal solution counts may lead to the *opposite* conclusion that mean best-of-fitness measures lead to. This begs a reevaluation of much of the GP literature, as published results may be dubious, and in some cases the opposite of their intended meaning.

Acknowledgments

Our thanks to Lee Spector, Paul Wiegand, and Ken De Jong for their insights and responses, and to our reviewers for their helpful comments.

References

Angeline, P. J. (1996). An investigation into the sensitivity of genetic programming to the frequency of leaf selection during subtree crossover. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 21–29, Stanford University, CA, USA. MIT Press.

Angeline, P. J. (1997). Subtree crossover: Building block engine or macromutation? In Koza, J. R., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M., Iba, H., and Riolo, R. L., editors, *Genetic Programming 1997: Proceedings*

of the Second Annual Conference, pages 9–17, Stanford University, CA, USA. Morgan Kaufmann.

Banzhaf, W., Nordin, P., Keller, R. E., and Francone, F. D. (1998). *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, dpunkt.verlag.

Daida, J., Ross, S., McClain, J., Ampy, D., and Holczer, M. (1997). Challenges with verification, repeatability, and meaningful comparisons in genetic programming. In Koza, J. R., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M., Iba, H., and Riolo, R. L., editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 64–69, Stanford University, CA, USA. Morgan Kaufmann.

Daida, J. M., Ampy, D. S., Ratanasavetavadhana, M., Li, H., and Chaudhri, O. A. (1999a). Challenges with verification, repeatability, and meaningful comparison in genetic programming: Gibson's magic. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M., and Smith, R. E., editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1851–1858, Orlando, Florida, USA. Morgan Kaufmann.

Daida, J. M., Yalcin, S. P., Litvak, P. M., Eickhoff, G. A., and Polito, J. A. (1999b). Of metaphors and darwinism: Deconstructing genetic programming's chimera. In Angeline, P. J., Michalewicz, Z., Schoenauer, M., Yao, X., and Zalzala, A., editors, *Proceedings of the Congress on Evolutionary Computation*, volume 1, pages 453–462, Mayflower Hotel, Washington D.C., USA. IEEE Press.

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

Langdon, W. B., Soule, T., Poli, R., and Foster, J. A. (1999). The evolution of size and shape. In Spector, L., Langdon, W. B., O'Reilly, U.-M., and Angeline, P. J., editors, *Advances in Genetic Programming 3*, chapter 8, pages 163–190. MIT Press, Cambridge, MA, USA.

Luke, S. (2000). *Issues in Scaling Genetic Programming: Breeding Strategies, Tree Generation, and Code Bloat*. PhD thesis, Department of Computer Science, University of Maryland, A. V. Williams Building, University of Maryland, College Park, MD 20742 USA.

Luke, S. (2001a). ECJ 7: An evolutionary computation research system in Java. Available at http://www.cs.umd.edu/projects/plus/ec/ecj/.

Luke, S. (2001b). When short runs beat long runs. In Spector, L., Goodman, E. D., Wu, A., Langdon, W. B., Voigt, H.-M., Gen, M., Sen, S., Dorigo, M., Pezeshk, S., Garzon, M. H., and Burke, E., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 74–80, San Francisco, California, USA. Morgan Kaufmann.

Luke, S. and Spector, L. (1997). A comparison of crossover and mutation in genetic programming. In Koza, J. R., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M., Iba, H., and Riolo, R. L., editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 240–248, Stanford University, CA, USA. Morgan Kaufmann.

Luke, S. and Spector, L. (1998). A revised comparison of crossover and mutation in genetic programming. In Koza, J. R., Banzhaf, W., Chellapilla, K., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M. H., Goldberg, D. E., Iba, H., and Riolo, R., editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 208–213, University of Wisconsin, Madison, Wisconsin, USA. Morgan Kaufmann.

Paterson, N. and Livesey, M. (2000). Performance comparison in genetic programming. In Whitley, D., editor, *Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference*, pages 253–260, Las Vegas, Nevada, USA.

Lexicographic Parsimony Pressure

Sean Luke
George Mason University
http://www.cs.gmu.edu/∼sean/

Liviu Panait
George Mason University
http://www.cs.gmu.edu/∼lpanait/

Abstract

We introduce a technique called lexicographic parsimony pressure, for controlling the significant growth of genetic programming trees during the course of an evolutionary computation run. Lexicographic parsimony pressure modifies selection to prefer smaller trees only when fitnesses are equal (or equal in rank). This technique is simple to implement and is not affected by specific differences in fitness values, but only by their relative ranking. In two experiments we show that lexicographic parsimony pressure reduces tree size while maintaining good fitness values, particularly when coupled with Koza-style maximum tree depth limits.

1 INTRODUCTION

Like many arbitrary-sized representations in evolutionary computation, genetic programming (GP) individuals tend to grow significantly in size when no code growth counteragents are applied. This growth is relatively independent of significant increases in fitness. The phenomenon, known in GP circles as *bloat*, is shaping up to be a major impediment to GP's scalability to more difficult problems which necessitate longer evolutionary runs.

The chief way bloat is controlled in GP is through the use of breeding restrictions stipulating the maximum depth of a GP parse tree individual. Lately other approaches have taken root, most popularly various forms of *parsimony pressure*, where the size of an individual is taken into consideration during selection. Parsimony pressure has to date taken two basic forms: parametric parsimony pressure, where an individual's size directly changes its fitness, and pareto parsimony pressure, where an individual's size is considered as an additional objective in a pareto optimization scheme.

In this paper we present a new family of parsimony pressure techniques which we think may be particularly apropos to GP and other evolutionary systems with large numbers of fitness-equivalent individuals in a population. This family is collectively known as *lexicographic parsimony pressure*, and is based on the idea of placing fitness, then size in lexicographic order; that is, preferring smaller individuals only when fitness is identical (or in some versions, similar). Lexicographic parsimony pressure is simple to implement, and it is less tied to the specific absolute fitness values in the population than parametric techniques are, much in the same way that tournament selection touted over fitness-proportionate selection.

We open the paper with discussions of current bloat-control techniques, followed by a description of lexicographic parsimony pressure and variations we have tried. We then give the results of an experiment showing that in most cases lexicographic parsimony pressure produces equivalent best-fitness-of-run results with significantly smaller trees than does depth restriction, except in the Symbolic Regression domain, where it performs poorly. We then give the results of a second experiment where we show that combinations of lexicographic parsimony pressure and depth limiting work very well compared to depth limiting alone.

2 CONTROLLING BLOAT

The evolutionary computation community has tried a number of approaches to controlling the growth of arbitrary-sized individuals. First and foremost are a number of parsimony pressure techniques, which include consideration of an individual's size as part of the selection procedure. Genetic programming has popularized some other techniques. Below we list four such techniques, followed by a range of parsimony pressure approaches.

Maximum Size or Depth Restriction This approach simply limits the maximum size of an individual, usually by rejecting large children as part of the breeding process.

For example, much work in GP follows the technique used in Koza [1992], which restricts modification operators to produce new trees of depth less than 17.

Explicitly Defined Introns This GP-specific technique allows the inclusion of special nodes which adapt the likelihood of subtree crossover or mutation at specific positions in the tree [Nordin et al. 1996].

Code Editing One easy way to attack growth is to directly simplify and optimize an individual's parse tree. Soule et al. [1996] for example report strong results with this approach. However, Haynes [1998] warns that editing can lead to premature convergence.

Pseudo-Hillclimbing This technique rejects children if they are not superior to (or simply different from) their parents in fitness. If a child is rejected from joining the next generation, a copy of its parent joins the next generation in its stead. One effect of this technique is to replicate large numbers of parents into future generations; earlier individuals are generally smaller than later individuals (hence the bloat), this results in slower growth in average size. This technique has been reported with some success in [Langdon and Poli 1998; Soule and Foster 1998b].

2.1 PARSIMONY PRESSURE

Unlike the techniques mentioned earlier, parsimony pressure is not GP-specific and has been used whenever arbitrary-sized representations tended to get out of hand. Such usage to date can be divided into two broad categories: parametric parsimony pressure, where size is a direct numerical factor in fitness, and pareto parsimony pressure, where size is considered as a separate objective in a pareto-optimization procedure.

Parametric Parsimony Pressure This includes size metrics, along with raw fitness, as part of an equation in computing the final fitness of an individual. For purposes of the discussion, let f be the individual's raw fitness, where higher is better, and g be the fitness after parsimony pressure is considered. Let s be an individual's size, and let a, b, c be arbitrary constants.

The most widely-used approach to parametric parsimony pressure is to treat the individual's size as a linear factor in fitness, that is, $g = af - bs$. This technique has been used in both GP [Koza 1992] and in non-GP [Burke et al. 1998]. Soule and Foster [1998a] present an interesting analysis of linear parsimony pressure and when and why it can fail. Linear parsimony pressure is occasionally augmented with a limit, that is if $s \quad c$ then $g = af$, else $g = af + b(c - s)$ [Cavaretta and Chellapilla 1999]. Belpaeme [1999] used a similar limit, but considered maximal tree depth rather than size as the parameter. Nordin and Banzhaf [1995] also applied parametric parsimony pressure, believed to be linear, to evolve machine language GP strings.

Linear parsimony pressure has been used in combination with adaptive strategies. Zhang and Mühlenbein [1995] adjusted b based on current population quality. Iba et al. [1994] propose a similar technique, except they use information-theoretic functions for f and s. Linear parsimony pressure has also been applied in stages: first by setting $g = f$, then factoring in size only after the population has reached a sufficient quality [Kalganova and Miller 1999]. Some non-GP papers [Wu et al. 1999; Bassett and De Jong 2000] use a nonlinear parsimony pressure: $g = (1 - as)f$. Bassett and De Jong note that this has the added feature of increasing the penalty proportionally to the fitness.

The problem with parametric parsimony pressure is exactly that: it is parametric, rather than based on rank. One must tune the parsimony pressure so as not to overwhelm the fitness metric. This can be difficult when the fitness assessment procedure is nonlinear, as is usually the case: it may well be that a difference between 0.9 and 0.91 in fitness is much more significant than a difference between 0.7 and 0.9. Parametric parsimony pressure can thus give size an unwanted advantage over fitness when the difference in fitness is only 0.01 as opposed to 0.2. Unexpected strength in the size parameter can also arise when the population's fitnesses are converging late in the evolution procedure. These issues are similar to those which gave rise to the preference of tournament selection and other nonparametric selection procedures over fitness-proportionate selection.

Pareto Parsimony Pressure The recent trend in parsimony pressure has been to treat it as a separate objective in a nonparametric, pareto optimization scheme. Pareto optimization is used when the evolutionary system must optimize for two or more objectives at once, but it is not clear which objective is "more important". An individual A is said to *pareto-dominate* another individual B if A is as good as B on all objectives, and better than B in at least one objective. One pareto optimization scheme assumes that one individual has a higher fitness than another if it dominates the other. Another scheme bases the fitness of individuals on the number of other individuals they dominate.

Pareto parsimony pressure treats raw fitness as one objective to optimize, and the individual's size as another objective. One particularly enticing feature of pareto parsimony pressure is that there is nothing to tune. Unfortunately, the technique has so far had mixed results in the literature. Some papers report smaller trees and the discovery of more

ideal solutions [Bleuler et al. 2001; DeJong et al. 2001], but tellingly they omit best-fitness-of-run results.[1] Ekart and Nemeth [2001] report the mean best-fitness-of-run, but it is *worse* than when not using the technique.

3 LEXICOGRAPHIC PARSIMONY PRESSURE

Lexicographic parsimony pressure is a straightforward multiobjective technique for optimizing both fitness and tree size, by treating fitness as the primary objective and tree size as a secondary objective in a lexicographic ordering. The procedure does not assign a new fitness value, but instead uses a modified tournament selection operator to consider size.

To select an individual, two individuals are chosen at random, and their fitnesses compared. If an individual has superior fitness, it is selected. If the fitnesses are the same, then sizes are compared, and the smaller individual is selected. If both fitness and size are the same, an individual is selected at random.

We think the procedure is attractive because it is based on the relative rank of individuals in a population rather than their explicit fitness values: thus specific differences in fitness are immaterial. All that matters is that one fitness is greater than another. Additionally, plain lexicographic parsimony pressure has nothing to tune. However, the procedure only works well in environments which have a large number of individuals with identical fitness. As it so happens, genetic programming is just such an environment, thanks to a large amount of inviable code (regions where crossover has no effect) and other events causing neutral crossovers and mutations.

Of course, there exist problem domains where few individuals have the same fitness. For these domains we propose two possible modifications of lexicographic parsimony pressure, both based on the notion of sorting the population, putting it into ranked buckets, and treating each individual in the bucket as if it had the same fitness. These two modifications are:

Direct Bucketing The number of buckets, b, is specified beforehand, and each is assigned a rank from 1 to b. The population, of size p, is sorted by fitness. The bottom $p\ b$ individuals are placed in the worst ranked bucket, plus any individuals remaining in the population with the same fitness as the best individual in the bucket. Then the second worst $p\ b$ individuals are placed in the second worst ranked bucket, plus any individuals in the population equal in fitness to the best individual in that bucket. This continues until there are no individuals in the population. Note that the topmost bucket with individuals can hold fewer than $p\ b$ individuals, if p is not a multiple of b. Depending on the number of equal-fitness individuals in the population, there can be some top buckets that are never filled. The fitness of each individual in a bucket is set to the rank of the bucket holding it. Direct bucketing has the effect of trading off fitness differences for size. Thus the larger the bucket, the stronger the emphasis on size as a secondary objective.

Ratio Bucketing Here the buckets are proportioned so that low-fitness individuals are placed into much larger buckets than high-fitness individuals. A bucket ratio $1\ r$ is specified beforehand. The bottom $1\ r$ fraction of individuals of the population are placed into the bottom bucket. If any individuals remain in the population with the same fitness as the best individual in the bottom bucket, they too are placed in that bucket. Of the remaining population, the next $1\ r$ fraction of individuals are placed into the next bucket, plus any individuals remaining in the population with the same fitness as the best individual now in that bucket, and so on. This continues until every member of the population has been placed in a bucket. Once again, the fitness of every individual in a bucket is set to the rank of the bucket relative to other buckets. As the remaining population decreases, the $1\ r$ fraction also decreases: hence higher-ranked buckets generally hold fewer individuals than lower-ranked buckets. Ratio bucketing thus allows parsimony to have more of an effect on average when two similar low-fitness individuals are considered than when two high-fitness individuals are considered.

Both bucketing schemes fill the buckets with remaining individuals equal in fitness to the best individual in the bucket. The purpose of this is to guarantee that all individuals of the same fitness fall into the same bucket and thus have the same rank. This removes artifacts due to the particular ordering of the population. Bucketing schemes require that the user specify a bucket parameter (either the number of buckets or the bucket ratio). This parameter guides how strong an effect parsimony can have on the selection procedure. Note however that this parameter is not a direct factor in fitness. Thus the specific difference in fitness between two individuals is still immaterial; all that matters is fitness rank.

We are aware of two papers in the literature which have used variations on lexicographic parsimony pressure. Lucas [1994] used a linear parametric function to evolve bitstrings used in context-free grammars: but the size was multiplied by a constant small enough to guarantee that the largest possible advantage for small size was less than

[1] As we argue in an accompanying paper, ideal-solution counts are a very poor measure of quality. Not only are they statistically invalid, but in fact are not correlated, or as badly as *inversely correlated*, with mean best-fitness-of-run results.

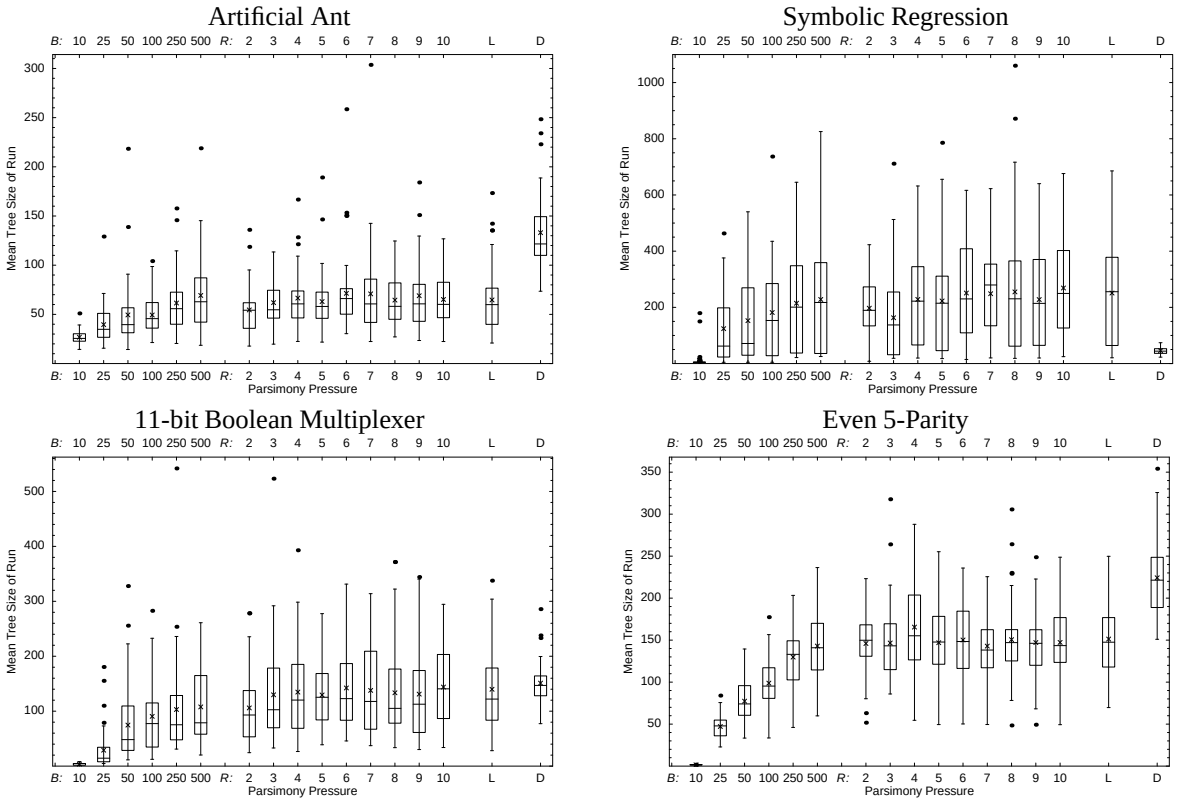


Figure 1: Boxplots of distributions of mean-tree-size-of-run for various parsimony pressure methods, as compared compared to plain depth limiting (labeled *D*). Lexicographic parsimony pressure is labeled *L*. Direct bucketing is labeled *B*, with the given number of buckets. Ratio bucketing is labeled *R* with the given ratio value.

the smallest difference in fitness. We believe the fitness was then fed into a fitness-proportional selection operator. In the *pygmies and civil servants* algorithm [Ryan 1994], crossover is always between one "civil servant" and one "pygmy". A pygmy is selected using linear parsimony pressure with a heavy weight for small size. A civil servant is selected using plain lexicographic selection. Both papers mention parsimony advantages only in passing.

4 EXPERIMENTS

Like most parsimony pressure literature, we chose to compare against the most popular technique for size restriction, namely Koza-style depth limiting. We performed two sets of experiments. The first experiment compared lexicographic parsimony pressure against depth limiting. The second experiment used lexicographic parsimony pressure in combination with depth limiting.

The experiments used population sizes of 1000. Without parsimony pressure, the depth ordered runs used plain tournament selection with a tournament size of 2. We chose four problem domains: Artificial Ant, 11-bit Boolean Multiplexer, Symbolic Regression, and Even-5 Parity. We followed the parameters specified in these four domains as set forth in Koza [1992]. Symbolic Regression used no ephemeral random constants. Artificial Ant used the Santa Fe food trail. Statistical significance was determined with ANOVAs at 95%. The evolutionary computation system used was ECJ 7 [Luke 2001].

As lexicographic ordering is influenced by likelihood of individuals having the same (or similar) fitness, it is useful to note the features of these four problem domains in this respect. Artificial Ant evolves trees to control an ant to eat as many food pellets as possible within 400 time steps. Fitness is simply the number of pellets, and the trail has only 89 of them, so there are relatively few fitness values an individual may take on. The 11-bit Boolean Multiplexer and Even-5 Parity problems both require the individual to learn a complex boolean function. 11-bit Boolean Multiplexer has integer fitness values ranging from 0 to 2048. It is known that 11-bit Boolean Multiplexer has relatively little inviable code, but most individuals' fitnesses fall into multiples of 32 or 64. Even-5 Parity has the fewest number of fitness values: only integer fitness values ranging

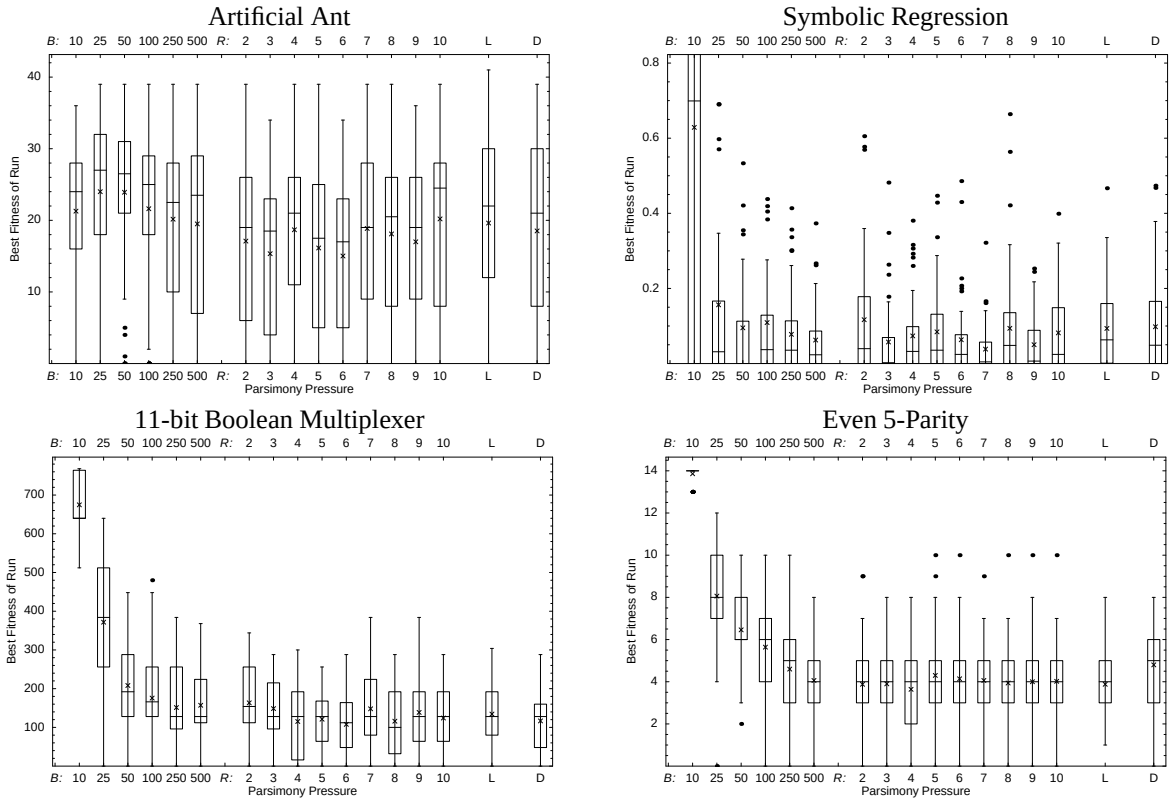


Figure 2: Boxplots of distributions of best-fitness-of-run for various parsimony pressure methods, as compared compared to plain depth limiting (labeled *D*). Lexicographic parsimony pressure is labeled *L*. Direct bucketing is labeled *B*, with the given number of buckets. Ratio bucketing is labeled *R* with the given ratio value. Lower fitness is better.

from 0 to 33. Symbolic Regression asks trees to fit a real-valued function within the domain [-1,1] but with any valid range; thus individuals can take on any real-valued fitness. However, Symbolic Regression suffers from a very large amount of inviable code, so many individuals in the population have the same fitness.

4.1 FIRST EXPERIMENT

The first experiment compared depth limiting against pure parsimony pressure approaches. Specifically, the techniques compared are:

Lexicographic parsimony pressure with direct bucketing, using 10, 25, 50, 100, 250, or 500 buckets.

Lexicographic parsimony pressure with ratio bucketing, using bucket ratios of 1/2, 1/3, 1/4, 1/5, 1/6, 1/7, 1/8, 1/9, or 1/10.

Plain lexicographic parsimony pressure.

Depth limiting (to 17).

We did 50 runs per technique, and plotted boxplots[2] showing the distribution of the best fitness per run, and also of the average tree size per run. Runs continued for 51 generations, and did not stop on the discovery of the optimum. Results are shown in Figures 1 and 2.

In the Artificial Ant and Even 5-Parity problems, all parsimony pressure techniques yielded statistically significantly superior tree size results to depth limiting, and had statistically insignificant differences in fitness, except for direct bucket numbers of 10, 25, and 50 for Even 5-Parity, which had worse fitness values. Small-numbered direct bucketing yielded much better tree sizes. For Even-5 Parity, this came at the cost of much worse fitness values. Artificial Ant, there was no difference in fitness.

For 11-bit Boolean Multiplexer, all parsimony pressure techniques had smaller mean tree sizes than depth limiting, but only direct bucketing numbers of 10, 25, 50, and 100 had statistically significant differences. Similarly, all

[2] In a boxplot, the rectangular region covers all values between the first and third quartiles, the stems mark the furthest individual within 1.5 of the quartile ranges, and the center horizontal line indicates the median. Dots show outliers, and $\times$ marks the mean.

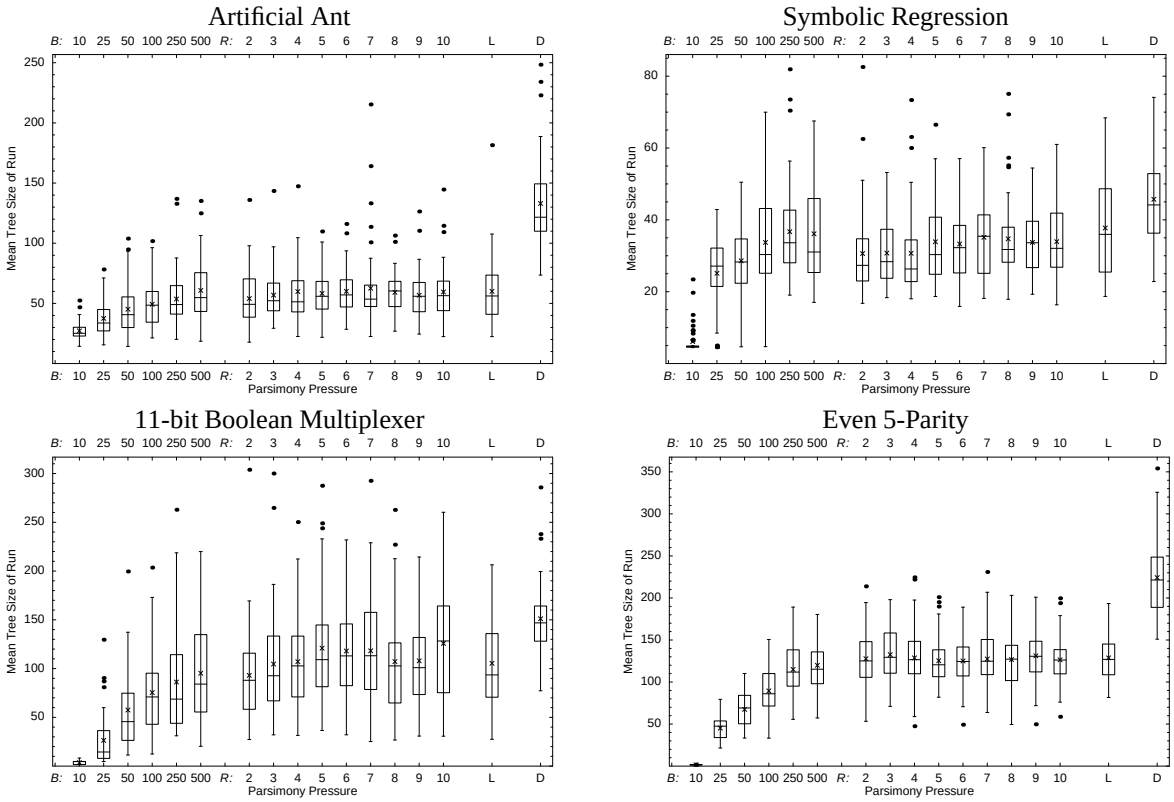


Figure 3: Boxplots of distributions of mean-tree-size-of-run for various parsimony pressure methods in combination with depth limiting, as compared compared to plain depth limiting (labeled *D*). Lexicographic parsimony pressure is labeled *L*. Direct bucketing is labeled *B*, with the given number of buckets. Ratio bucketing is labeled *R* with the given ratio value.

techniques had statistically insignificant differences in fitness except for direct bucking numbers of 10, 25, and 50, which had worse fitness.

The surprise came with Symbolic Regression. We had expected lexicographic parsimony pressure to yield poor tree sizes in this domain relative to depth limiting, and it did. But interestingly, bucketing also had poor tree sizes. Only direct bucking with 10 buckets yielded statistically significantly worse fitness than depth limiting.

Growth Curves For the Even-5 Parity problem, parsimony pressure techniques generally held tree growth to a standstill or began lowering tree sizes it by generation 30. For Artificial Ant, this occurred by about generation 10. In 11-bit Boolean Multiplexer, generation 40; most parsimony pressure techniques were lowering tree sizes by then as well. In Symbolic Regression, tree growth for all the parsimony pressure techniques rose in a quadratic curve similar to that found for unrestricted GP in this problem. With depth limiting in all four problem domains, mean tree growth continued to rise linearly.

Lexicographic parsimony pressure has an Achilles' heel: if GP can create incrementally better trees by tacking subtrees onto their periphery, then lexicographic parsimony pressure cannot act against it. As long as the trees are infinitesimally better, size does not come into play. Symbolic Regression has this property, and we had expected plain lexicographic parsimony pressure to do badly in this domain. But we were very surprised to see the poor performance of bucketing approaches as well.

4.2 SECOND EXPERIMENT

If depth limiting did well compared to lexicographic parsimony pressure in Symbolic Regression, and held its own reasonably in 11-bit Boolean Multiplexer, we wondered how the combination of the two techniques would fare. Our second experiment compared the same techniques as in the first experiment, but combined the parsimony pressure techniques with depth limiting. Again, we did 50 runs per technique. These results are shown in Figures 3 and 4.

This time, parsimony pressure plus depth limiting significantly outperformed depth limiting alone. In the Symbolic Regression, Artificial Ant, and Even-5 Parity problems, all

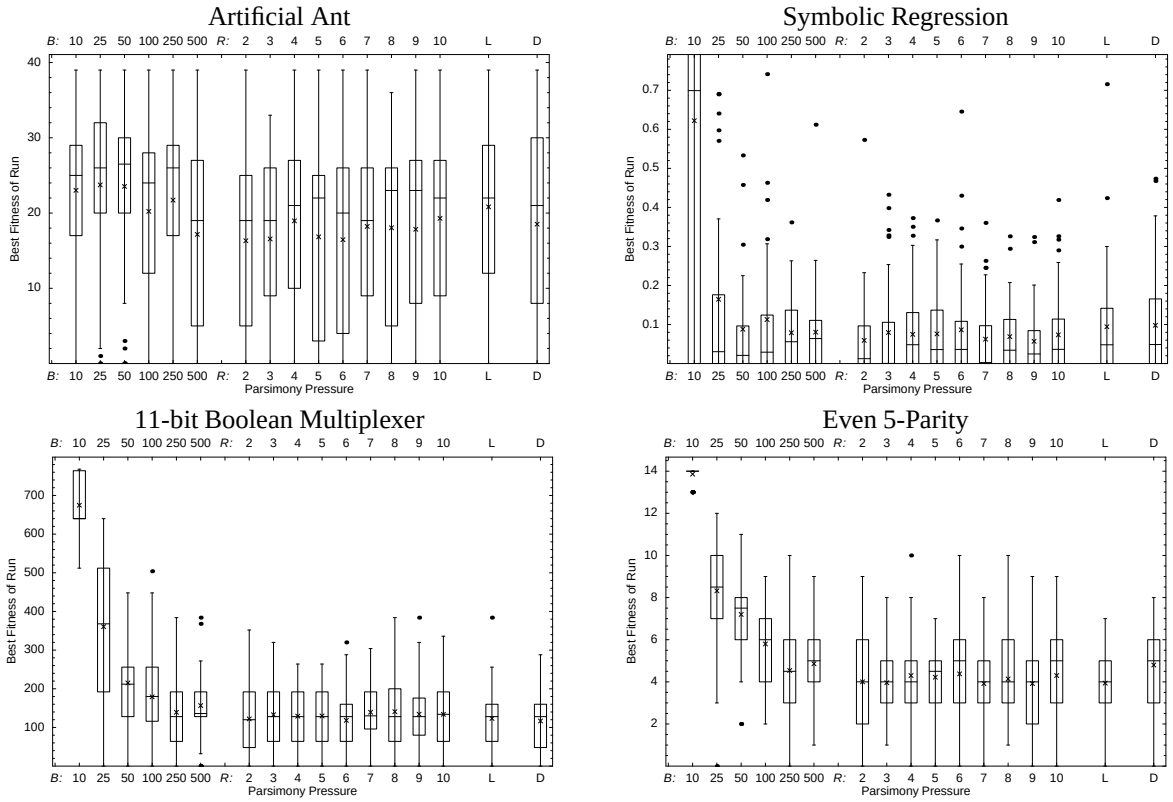


Figure 4: Boxplots of distributions of best-fitness-of-run for various parsimony pressure methods in combination with depth limiting, as compared compared to plain depth limiting (labeled *D*). Lexicographic parsimony pressure is labeled *L*. Direct bucketing is labeled *B*, with the given number of buckets. Ratio bucketing is labeled *R* with the given ratio value. Lower fitness is better.

applications of parsimony pressure plus depth limiting had statistically significantly superior tree sizes when compared to plain depth limiting. In the 11-bit Boolean Multiplexer, this was also the case except for ratio buckets of size 1/5, 1/7, and 1/10, which had statistically insignificant differences with depth limiting.

As before, there were no statistically significant differences in fitness in the Artificial Ant problem. Direct bucketing with 10, 25, and 50 buckets yielded statistically significantly worse fitness than depth limiting for the Even-5 Parity and 11-bit Boolean Multiplexer problems. In the Symbolic Regression problem, only direct bucketing with 10 buckets had statistically worse fitness than depth limiting.

Growth Curves In the Symbolic Regression and Even-5 Parity problems, parsimony pressure plus depth limiting flattened out tree growth by about generation 25. In the Artificial Ant problem, parsimony pressure plus depth limiting dropped sizes after about generation 5, flattening out at about generation 20. In the 11-bit Boolean Multiplexer, the same techniques began slowly lowering tree sizes at about generation 35.

5 CONCLUSIONS AND FUTURE WORK

In three of four problem domains, lexicographic parsimony pressure and its variants (direct bucketing and ratio bucketing, given reasonable parameter values) maintained the same mean best-fitness-of-run as did Koza-style depth limiting, with equivalent or significantly lower mean tree sizes. But in Symbolic Regression, where incrementally larger trees are often (just barely) superior in fitness, lexicographic techniques were practically helpless to stop bloat. However, a combination of depth limiting and lexicographic parsimony pressure consistently outperformed depth limiting in capping bloat, while maintaining statistically equivalent mean best-fitness-of-run values. Given its simple implementation and general applicability, we hope lexicographic parsimony pressure may prove a popular approach to bloat control. We plan to extend this work to other techniques such as layered tournaments which alternately consider fitness and size. We also plan to compare directly to parametric parsimony pressure and pareto-optimization-based methods in the future.

References

Bassett, J. K. and De Jong, K. A. (2000). Evolving behaviors for cooperating agents. In *International Syposium on Methodologies for Intelligent Systems*, pages 157–165.

Belpaeme, T. (1999). Evolution of visual feature detectors. In Poli, R., Cagnoni, S., Voigt, H.-M., Fogarty, T., and Nordin, P., editors, *Late Breaking Papers at EvoISAP'99: the First European Workshop on Evolutionary Computation in Image Analysis and Signal Processing*, pages 1–10, Goteborg, Sweden.

Bleuler, S., Brack, M., Thiele, L., and Zitzler, E. (2001). Multiobjective genetic programming: Reducing bloat using spea2. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*, pages 536–543, COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea. IEEE Press.

Burke, D. S., De Jong, K. A., Grefenstette, J. J., Ramsey, C. L., and Wu, A. S. (1998). Putting more genetics into genetic algorithms. *Evolutionary Computation*, 6(4):387–410.

Cavaretta, M. J. and Chellapilla, K. (1999). Data mining using genetic programming: The implications of parsimony on generalization error. In Angeline, P. J., Michalewicz, Z., Schoenauer, M., Yao, X., and Zalzala, A., editors, *Proceedings of the Congress on Evolutionary Computation*, volume 2, pages 1330–1337, Mayflower Hotel, Washington D.C., USA. IEEE Press.

DeJong, E. D., Watson, R. A., and Pollack, J. B. (2001). Reducing bloat and promoting diversity using multi-objective methods. In Spector, L., Goodman, E. D., Wu, A., Langdon, W. B., Voigt, H.-M., Gen, M., Sen, S., Dorigo, M., Pezeshk, S., Garzon, M. H., and Burke, E., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 11–18, San Francisco, California, USA. Morgan Kaufmann.

Ekart, A. and Nemeth, S. Z. (2001). Selection based on the pareto nondomination criterion for controlling code growth in genetic programming. *Genetic Programming and Evolvable Machines*, 2(1):61–73.

Haynes, T. (1998). Collective adaptation: The exchange of coding segments. *Evolutionary Computation*, 6(4):311–338.

Iba, H., de Garis, H., and Sato, T. (1994). Genetic programming using a minimum description length principle. In Kinnear, Jr., K. E., editor, *Advances in Genetic Programming*, chapter 12, pages 265–284. MIT Press.

Kalganova, T. and Miller, J. (1999). Evolving more efficient digital circuits by allowing circuit layout evolution and multi-objective fitness. In Stoica, A., Keymeulen, D., and Lohn, J., editors, *Proceedings of the 1st NASA/DoD Workshop on Evolvable Hardware (EH'99)*, pages 54–63, Piscataway, NJ. IEEE.

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

Langdon, W. B. and Poli, R. (1998). Genetic programming bloat with dynamic fitness. In Banzhaf, W., Poli, R., Schoenauer, M., and Fogarty, T. C., editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 96–112, Paris. Springer-Verlag.

Lucas, S. (1994). Structuring chromosomes for context-free grammar evolution. In *Proceedings of the First IEEE Conference on Evolutionary Computation*, pages 130–135. IEEE.

Luke, S. (2001). ECJ 7: An EC and GP system in Java. http://www.cs.umd.edu/projects/plus/ec/ecj/.

Nordin, P. and Banzhaf, W. (1995). Complexity compression and evolution. In Eshelman, L., editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 310–317, Pittsburgh, PA, USA. Morgan Kaufmann.

Nordin, P., Francone, F., and Banzhaf, W. (1996). Explicitly defined introns and destructive crossover in genetic programming. In Angeline, P. J. and Kinnear, Jr., K. E., editors, *Advances in Genetic Programming 2*, pages 111–134. MIT Press, Cambridge, MA, USA.

Ryan, C. (1994). Pygmies and civil servants. In Kinnear, Jr., K. E., editor, *Advances in Genetic Programming*, chapter 11, pages 243–263. MIT Press.

Soule, T. and Foster, J. A. (1998a). Effects of code growth and parsimony pressure on populations in genetic programming. *Evolutionary Computation*, 6(4):293–309.

Soule, T. and Foster, J. A. (1998b). Removal bias: a new cause of code growth in tree based evolutionary programming. In *1998 IEEE International Conference on Evolutionary Computation*, pages 781–186, Anchorage, Alaska, USA. IEEE Press.

Soule, T., Foster, J. A., and Dickinson, J. (1996). Code growth in genetic programming. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 215–223, Stanford University, CA, USA. MIT Press.

Wu, A., Schultz, A., and Agah, A. (1999). Evolving control for distributed micro air vehicles. In *IEEE Computational Intelligence in Robotics and Automation Engineers Conference*.

Zhang, B.-T. and Mühlenbein, H. (1995). Balancing accuracy and parsimony in genetic programming. *Evolutionary Computation*, 3(1):17–38.

An Analysis of Random Number Generators for a Hardware Implementation of Genetic Programming using FPGAs and Handel-C

Peter Martin
Department of Computer Science, Essex University,
Colchester, Essex, UK
petemartin@ntlworld.com

Abstract

This paper analyses the effect of using different random number generators (RNG) in a hardware implementation of Genetic Programming using Field Programmable Gate Arrays. Hardware systems have typically used RNGs based on Logical Feedback Shift Registers or Cellular Automata. Different configurations of these generators are evaluated as well as using a source of true random numbers and a standard multiply/add generator. The results show that using a more sophisticated generator than a simple LFSR slightly improves the performance of GP.

1 Introduction

Previous work [11] described an implementation of Genetic Programming using a Field Programmable Gate Array (FPGA) and a high level language to hardware compilation system called Handel-C. Subsequent work [12] described a pipelined implementation that improved the performance and demonstrated that the technique could be used to solve the artificial ant problem . In both cases the work concentrated on the implementation issues and improving the clock speed of the implementation, but put to one side the performance of the system with respect to its ability to solve GP problems. Now that the raw throughput issues have been addressed it is time to look at how good the hardware implementation performs, in particular the effectiveness of the Random Number Generator (RNG) used.

A comment often made about Genetic Programming and other stochastic search methods is that a good random number generator is needed. The evidence so far is that the quality of the RNG is probably not as important as often stated. Nevertheless, it is important to consider the effect of design decisions and to investigate alternatives where practicable.

In the hardware implementation of GP, the random number generator is implemented using a Logical Feedback Shift Register (LFSR) which has a number of known weaknesses. This suggests that other random number generators should be investigated. This paper begins with a brief description of the hardware GP system and Handel-C. This is followed by a review of previous work on random number generation that has been implemented in hardware. We then present an analysis of the pseudo random number generator used in the original design, and investigate other random number generators. We finish with a discussion of the results and draw some conclusions.

2 A Hardware Implementation of GP using FPGAs

Implementing GP in hardware is motivated by the potential speedups that can be obtained. The platform chosen is an FPGA which is a reconfigurable logic circuit than can be programmed to perform a wide range of logic functions. A typical FPGA is arranged as an array of configurable logic cells, input-output circuits and programmable interconnections. A typical FPGA architecture is shown in Figure 1.

Traditionally FPGAs have been programmed using hardware design languages such as VHDL[1], but an alternative approach using high level language to hardware compilation techniques has been developed, which allows a high level imperative language to be used to generate the configuration information for the FPGA. Handel-C is one example of this technology, and has been used for the work described in this paper.

[1] VHDL is a standard hardware design language. It stands for VHSIC Hardware Design Language. VHSIC itself stands for Very High Speed Integrated Circuit.

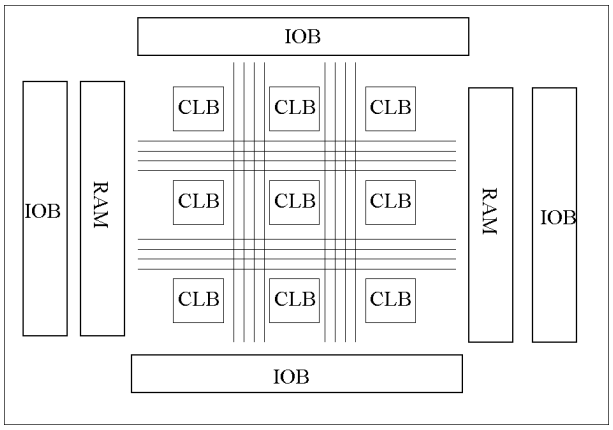


Figure 1: Typical FPGA architecture. The CLBs are the configurable logic blocks, IOBs are the Input Output Blocks and the RAMs are on-chip Random Access memory blocks.

Handel-C is a high level language that is at the heart of a hardware compilation system known as Celoxica DK1 [4] which is designed to compile programs written in a C-like high level language into synchronous hardware. The output from Handel-C is a file that is used to create the configuration data for the FPGA. A description of the process used by Handel-C to transform a high level language into hardware and examples of the hardware generated can be found in [19]. The C-like syntax makes the tool appealing to software engineers with little or no experience of hardware. They can quickly translate a software algorithm into hardware, without having to learn about VHDL or FPGAs in detail.

2.1 Target Hardware

The target hardware for this work is a Celoxica RC1000 FPGA development board fitted with a Xilinx XCV2000E Virtex-E FPGA having 43,200 logic cells and 655,360 bits of block ram. The board also has a PCI bridge that communicates between the RC1000 board and the host computer's PCI bus, and four banks of Static Random Access Memory (SRAM). Fast switches isolate the FPGA from the SRAM, allowing both the host CPU and the FPGA to access the SRAM, though not concurrently.

2.2 Program Representation

The lack of a stack in Handel-C means that a standard tree based representation is difficult to implement because recursion cannot be handled by the language. An alternative to a tree representation is a linear representation which has been used by others to solve some hard GP problems [18]. Using a linear representation, a program consists of a sequence of words which are decoded by the problem specific fitness function.

2.3 Previous work using FPGAs in Evolutionary Computing

A detailed review of previous work using FPGAs in Evolutionary Computing can be found in [11].

3 Previous Work on Pseudo Random Numbers for Genetic Programming and Hardware

This section reviews the types of random number generators that have been used by hardware implementations of GA, GP and other applications of hardware to probabilistic algorithms.

Linear Feedback Shift Register (LFSR) or Tauseworth generators have been used by Maruyama et al [14]. In their paper they referred to the generator as a m-sequence, or maximal sequence. This means that the generator of length n generates $2^n - 1$ numbers. Graham [5] implemented a single cycle LFSR.

An interesting hybrid approach was used by Tommiska and Vuori [23] where three coupled LFSRs were used to provide a random sequence. An interesting feature of this work is that the RNG was combined with a source of noise. The amplified noise from a diode was fed into an analogue to digital converter, and the resulting digital values were used to seed the RNG, and also added to the LFSR at intervals.

The manufacturers of FPGAs provide example designs of LFSRs to be used as random sequence generators. For example Xilinx [25], and Altera [2] provide Hardware Design Language (HDL) code for LFSRs.

Aporntewan [3] used a one dimensional 2-state Cellular Automata (CA). Shackleford et al [21] implemented a CA based on the work by Wolfram [24].

In the field of GP, the behavior of GP and GAs has been investigated using different RNGs. Meysenburg and Foster considered the effect of different RNGs on GAs [16] and GP [15]. Their conclusions were that there were no statistically significant differences in the performance of GA or GP when different RNGs were used.

4 Experimental setup

The performance of the various RNGs was evaluated using three methods. Firstly, the Diehard test suite maintained by Marsaglia [10] was used to gauge the general perfor-

mance of the RNG. This suite consists of up to 15 tests that are modeled on applications of random numbers. All the RNGs considered in this paper were implemented in ISO-C and were submitted to all 15 tests. The test method for Diehard is similar to that described in Meysenburg and Foster [15]. Each RNG was used to generate a binary file of about 10 MiB[2]. Each Diehard test produces one or more p-values. A p-value can be considered good, bad, or suspect. Meysenburg used a scheme by Johnson [6] which assigns a score to a p-value as follows. If $p \geq 0.998$ then it is classified as bad. If $0.95 \leq p < 0.998$ then it is classified as suspect. All other p-values are classified as good. Every bad p-value scores 4, every suspect p-value scores 2 and good p-values score zero. For each RNG, the scores for each test were summed, and the total for each RNG is the sum of all the test scores for that RNG. Using this scheme, high scores indicate a poor RNG and low scores indicate a good RNG. The results for each test are given in Appendix A.

Each RNG was then implemented using Handel-C and used in the hardware implementation of the artificial ant problem [8][12]. In the hardware implementation the function set differs from the standard example in only having two functions: $\mathcal{F} = \{IF\_FOOD, PROGN2\}$ where IF_FOOD is a two argument function that looks at the cell ahead and if it contains food it evaluates the first terminal, otherwise it evaluates the second terminal. $PROGN2$ evaluates its first and second terminals in sequence. The terminal set $\mathcal{T} = \{LEFT, RIGHT, MOVE, NOP\}$, where $LEFT$ and $RIGHT$ change the direction the ant is facing, $MOVE$ moves the ant one space forwards to a new cell, and if the new cell contains food, the food is eaten. NOP is a no-operation terminal and has no effect on the ant but is included to make the number of terminals a power of 2, which simplifies the hardware logic. Each time $LEFT$, $RIGHT$ or $MOVE$ is executed, the ant consumes one time step. The run stops when either all the time steps have been used, or the ant has eaten all the food. All the experiments use the Santa Fe trail, which has 89 pellets of food. Each experiment was run 500 times and the total number of 100% correct programs recorded. This is used as a measure of how well the RNG performs. In all cases the population size is 1024, the maximum program length is 31 and all experiments were run for 31 generations. The ant was allocated 600 timesteps. The probability of selecting crossover was 67%, mutation 10% and reproduction 23%. The crossover operator used the truncating method of limiting the maximum program length, as described in [13].

Each RNG was also implemented as a stand alone application for an FPGA using Handel-C, and the number of slices used and the maximum attainable clock frequency was recorded. This gives a measure of the hardware resources needed to implement the RNG, and also an indication of the logic depth required.

[2]The notation MiB indicates 2^{20} (1048576) bytes. This paper uses the binary prefixes from the NIST.[17]

5 Random Number Generator Implementations

5.1 LFSR RNG

Figure 2 shows a schematic of the LFSR used in this work.

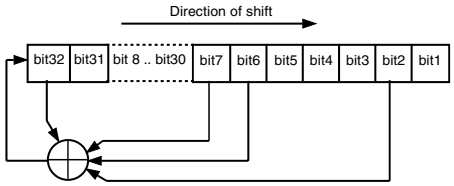


Figure 2: Logical Feedback Shift Register Random Number Generator

The random number is read from the highest bits as required. The obvious weakness of this type of RNG is that sequential values fail the serial test described by Knuth [7, pp 55-56]. At any time step t there is a 50% probability that the value at time $t+1$ can be predicted. If for an LFSR of length n at time t the value is v, then at time $t+1$ the value will be $v/2$ or $v/2+2^{n-1}$. This is shown in Figure 3 where pairs of values v_t and v_{t+1} are plotted.

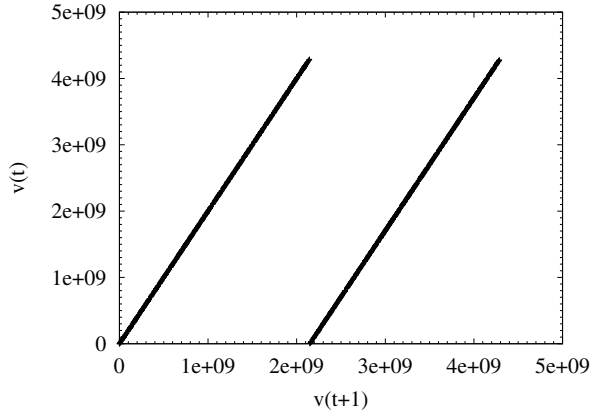


Figure 3: Serial test of a simple LFSR RNG

It can be seen that for any value v_t there are only two possible values of v_{t+1}. Though the random number generator runs in parallel with the main GP machine, it is possible to access sequential values when creating an initial program, or when choosing crossover points. There is then a possibility of a potentially degrading bias by using such an RNG.

5.2 Multiple LFSRs

One method of obtaining better serial test results for the LFSR of length n is to allow the LFSR to run for n cycles before reading another number. Since this would limit the rate at which random numbers could be generated in the present design it is not explored any further. However, an equivalent result can be obtained by implementing n LFSRs of length m and using a single bit from each LFSR at each time step. This can also be done using a single long LFSR of $n \times m$ bits, [22] effectively implementing n parallel LFSRs. However, implementing a long shift register in a Xilinx Virtex FPGA is not efficient because the look up tables can implement a 16 bit shift register very easily, but longer shift registers require more extensive routing resources.

The effect of using a better RNG was investigated by implementing 32 16 bit LFSR machines that run in parallel, and initializing each LFSR to a different value. Bit32 from each LFSR is used to construct a 32 bit random number. The serial test result is shown in Figure 4, which shows the serial test result for 32 LFSRs is better than the single LFSR. This generator is referred to as the 32LFSR.

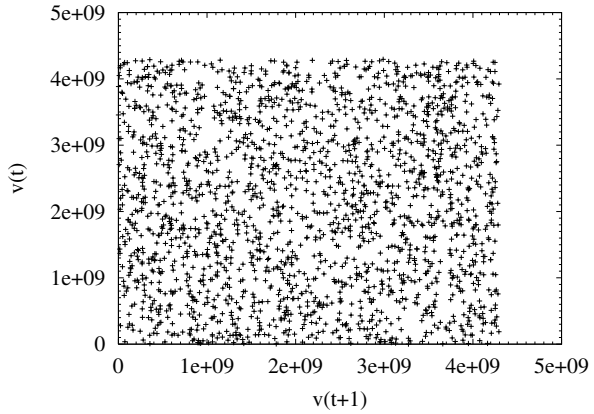


Figure 4: Serial test for an RNG using 32 parallel LFSRs

5.3 Cellular Automata RNG

Another popular RNG for hardware implementations is based on Cellular Automata (CA). A one-dimensional (1D) CA consists of a string of cells. Each cell has two neighbors - left and right, or in some literature west and east respectively. At each time step, the value of any cell c is given by a rule. For this implementation, rule 30 is used, which states that for any cell c at time t, $c_{t+1} = ((west_t + c_t) \oplus east_t)$, where $\oplus$ denotes the exclusive OR function. In practice the CA is implemented using a single 32 bit word, and for cell 0, its right-hand neighbor is cell 31, and similarly for cell 31 its left hand neighbor is cell 0. Figure 5 shows the result of running this RNG using the serial test. As in the simple LFSR RNG there is a distinct pattern to the numbers, but for most values of v_t there are several possible values for v_{t+1}. This generator is referred to as 1DCA.

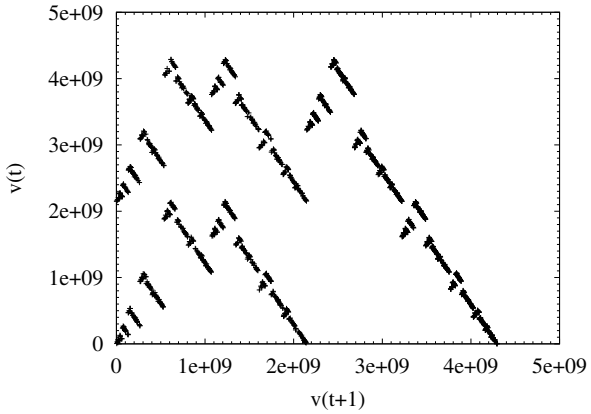


Figure 5: Serial test for a 1DCA RNG

5.4 Multiple CA generators

As in the case of the LFSR RNG, if several CAs are combined, the results should be much better. For this test, 32 CAs were implemented, and by taking one bit from each CA, a 32 bit random number can be generated. The serial test appears to be much more random, as shown in Figure 6. Each CA is initialized with a different pattern. This generator is referred to as the 32CA

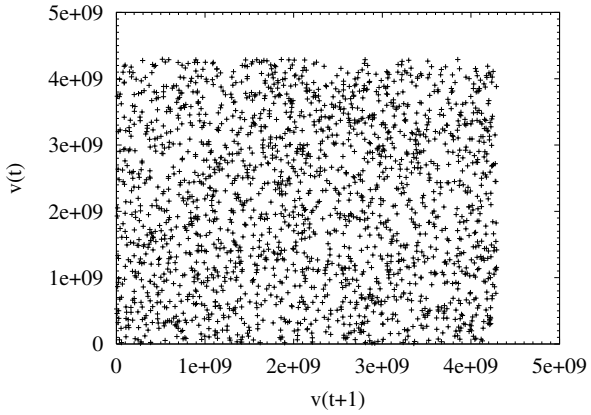


Figure 6: Serial test for a 32CA

5.5 Standard C RNGs

Another frequently used RNG is the linear congruential (LC) generator that is often found in implementations of the standard C library. The general equation for these is

$I_{j+1} = (aI_j + c) \bmod m$, where a, c and m are constants chosen to produce a maximal length RNG. However, as pointed out by many authors (eg:[20]) these generators are not good. Another factor against such a generator for implementing in hardware is that it requires one addition, one multiplication, and one modulus operator, which in Handel-C would consume a large amount of silicon and because of the deep logic produced, would be slow. An alternative given by [20] avoids the modulus operator, and is called the Even Quicker Generator (EQG). It is claimed that this is about as good as any 32 bit linear congruential generator. Its equation is $I_{j+1} = aI_j + c$, and values for $a = 1664525$ and $c = 1013904223$ are suggested.

As a sanity check that the experimental method of ranking the RNGs using Diehard was the same as that used by Meysenburg, the generator known as "the mother of all generators" was also implemented and run against the Diehard suite. This is a multiply with carry generator and is described by Marsaglia [9]. It was not implemented in the hardware GP system.

5.6 Non random sequences

Until now we have considered pseudo random sequences. These are sequences where it is hard to guess the next number in a sequence. As an experiment, a further set of runs were performed with an obviously non-random number generator. For this a sequential generator which generates the sequence $n, n+1, n+2, \ldots$ was used. Rather surprisingly this also worked to produce 100% correct programs, though substantially fewer than the other generators achieved.

5.7 Truly Random Sequences

All the RNGs considered so far are not true random sequences, relying on the manipulation of objects of finite size, and so fail one or more of the Diehard battery of tests. So a set of random numbers was obtained from a source generated by using the atmospheric noise captured by a radio receiver[1]. Each GP run for the ant problem needs about half a million random numbers, so a block of 10 MiB was downloaded from www.random.org, and a randomly selected 2 MiB block was transferred to one of the SRAM on the FPGA system using DMA. The FPGA read this block sequentially to get its random numbers.

As reported in [23], RNGs based on sampling a source of noise are often slow, so they are not always applicable to high speed systems.

6 Experimental Results

The results from running the Diehard tests are given in Appendix A and are summarized in Table 1. This shows the total results for each test and ranks them according to the Diehard score.

Table 1: Summary results of running the Diehard tests on the RNGS.

RNG	Score
Mother	20
True	22
32LFSR	162
EQG	288
32CA	640
CA	676
LFSR	756

The number of correct programs that were produced by running the ant problem on the hardware using each random number generator was recorded and is shown in Table 2. The results are ranked according to how many correct programs were found and shows how each RNG performed. The table also shows the slice count for the RNG implemented using Handel-C and the maximum clock rate as reported by the place and route tools. The slice count is a vendor and device dependent measure of the number of FPGA logic blocks that have been used. The clock rate is an indication of the logic depth required to implement the generator, with deeper logic having a greater gate delay, and therefore a lower maximum clock rate. The slice count and clock rate for the true RNG assumes that the source of random numbers is supplied by an external device to the FPGA, and that the FPGA simply reads the value from a port and writes it to a register.

Table 2: Summary of GP performance for all random number generators tested from 500 runs of the artificial ant problem

RNG	Rank	Correct	Slice	Clock rate F_{max} (MHz)
32CA	1	82	284	105
True	2	81	6	>200
32LFSR	3	79	130	134
EQG	4	78	288	42
ID CA	5	78	22	125
LFSR	6	68	18	188
Sequential	7	39	21	155

7 Discussion

The score obtained by the Mother RNG was close to that obtained by Meysenburg (19), the difference being explained by the fact that Meysenburg used the average of 32 runs using 32 different seeds, while the work described here used only a single run. It is likely that using 32 different seeds, that different scores would be observed. This confirms that the experimental method used for ranking the RNGs using Diehard is comparable.

Despite the apparently serious deficiencies found in both the simple LFSR used in the original implementation and the simple one dimensional CA random number generator, the overall effect of implementing a more sophisticated RNG on the overall GP performance appeared to be small. This result generally agrees with the work by Meysenburg and Foster [15], with the exception that they did not consider a single-cycle LFSR or an obviously non-random generator. The single-cycle LFSR performs the least well of the RNGs considered in this paper.

A surprising result was the emergence of programs when a non-random sequence was used. Clearly a non-random sequence does not allow GP to operate as efficiently in terms of producing 100% correct programs, presumably because of the failure to explore some areas of the search space.

Despite the small differences in performance, from the results we can say that using a different RNG from the single LFSR would improve the performance of the hardware GP implementation by a measurable and therefore useful amount, and that an RNG based on multiple LFSRs or multiple CAs would be a better choice for a hardware GP system. The use of a truly random number source did not appear to improve performance over the 1DCA, 32CA and 32LFSR RNGs. This provides more evidence countering the notion that GP needs a very high quality RNG.

Table 2 shows that the difference in GP performance between the 32CA, True, 32LFSR, EQG and 1DCA generators is small. However, these 5 generators have very different Diehard scores, so there does not appear to be a straightforward relationship between the Diehard score and the performance of GP. This raises a question about the role that RNGs play in GP. Is a RNG that scores well in the standard tests for randomness the best RNG for GP?

When looking at the FPGA slice counts and maximum clock rates, it is clear that the 32LFSR uses about half the FPGA resources of the 32CA, and the 32LFSR exhibits a smaller delay than the 32CA. As predicted, the EQG uses the most FPGA resources and has very deep logic, meaning that it can only run at a much slower rate than the other generators. The EQG RNG could be re-implemented in the FPGA using pipelines to achieve a higher clock rate, but since it performed no better than the 32CA and 32LFSR, this was not investigated any further.

Random numbers are used in several functions within a GP system: Initial population creation, selection and crossover point selection. In common with all reported GP systems, the same RNG is been used for all these functions within a run. From a practical point of view it would appear that there is little point in using more than one type of RNG for different functions, but from the result using a non-random sequence a question arises about the role that random sequences play as opposed to sequences that simply enumerate a set of numbers. From this it follows that different stages in GP may use random number sequences in different ways, and that using an enumeration may be helpful when investigating the dynamics of GP.

8 Conclusions

The main conclusion from this investigation is that for the hardware GP system, the simple LFSR used in the original design can be improved upon by using a generator based on multiple LFSRs, multiple CAs, or if available, a high speed source of true random numbers. A secondary conclusion is that with the exception of the non-random sequence and the single LFSR, there is no significant difference in GP performance when different hardware RNGs are used.

Acknowledgments

I would like to thank Riccardo Poli for his invaluable help in preparing this paper and the anonymous reviewers for their helpful comments. I would also like to thank Marconi plc, Celoxica Ltd. and Xilinx Inc. for supporting this work.

References

[1] random.org A source of true random numbers derived from radio noise and available on the internet. www.random.org, 2002. Last visited Jan 2 2002.

[2] Altera. Linear feedback shift register megafunction. http://www.altera.com/literature/sb/sb11_01.pdf, Dec. 2001.

[3] C. Aporntewan and P. Chongstitvatana. A Hardware Implementation of the compact genetic algorithm. *IEEE Congress on Evolutionary Computation*, pages 624–629, May 2001.

[4] Celoxica. Web site of Celoxica Ltd. www.celoxica.com, 2001. Vendors of Handel-C. Last visited 15/June/2001.

[5] P. Graham and B. Nelson. Genetic algorithms in software and in hardware - a performance analysis of

workstation and custom computing machine implementations. In K. Pocek and J. Arnold, editors, *Proceedings of the Fourth IEEE Symposium of FPGAs for Custom Computing Machines.*, pages 216–225, Napa Valley, Califormia, Apr. 1996. IEEE Computer Society Press.

[6] B. Johnson. Radix-b extensions to some common empirical tests for pseudorandom number generators. *ACM Transactions on Modelling and Computer Simulation*, 6(4):261–273, 1996.

[7] E. Knuth, Donald. *Semi numerical algorithms*, volume 2. Addison-Wesley Publishing Company, 1969.

[8] J. Koza. *Genetic programming: on the programming of computers by means of natural selection*. MIT Press, Cambridge, MA, USA, 1992.

[9] G. Marsaglia. Yet another RNG. Posted to sci.stat.math, 1 Aug. 1994.

[10] G. Marsaglia. Web site for Diehard random number test suite. http://stat.fsu.edu/geo/, 2001. Last visited 15/June/2001.

[11] P. Martin. A Hardware Implementation of a Genetic Programming System using FPGAs and Handel-C. *Genetic Programming and Evolvable Machines*, 2(4):317–343, 2001.

[12] P. Martin. A pipelined hardware implementation of genetic programming using FPGAs and Handel-C. In *Eurogp2002*, 2002.

[13] P. Martin and R. Poli. Crossover operators for a hardware implementation of genetic programming using FPGAs and Handel-C. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2002*, July 2002.

[14] T. Maruyama, T. Funatsu, M. Seki, Y. Yamaguchi, and T. Hoshino. A Field-Programmable Gate-Array system for Evolutionary Computation. *IPSJ Journal*, 40(5), 1999.

[15] M. Meysenburg and J. Foster. Random generator quality and GP performance. In W. Banzhaf, J. Daida, A. Eiben, M. Garzon, V. Honavar, M. Jakiela, and R. Smith, editors, *Proceedings of the genetic and evolutionary computation conference*, volume 2, pages 1121–1126, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[16] M. Meysenburg and J. Foster. Randomness and GA performance, revisited. In W. Banzhaf, J. Daida, A. Eiben, M. Garzon, V. Honavar, M. Jakiela, and R. Smith, editors, *Proceedings of the genetic and evolutionary computation conference*, volume 1, pages 425–432, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[17] NIST. The NIST reference on constants, units and uncertainty. http://physics.nist.gov/cuu/, 2002.

[18] P. Nordin and W. Banzhaf. Evolving turing-complete programs for a register machine with self-modifying code. In L. Eshelman, editor, *Genetic algorithms: proceedings of the sixth international conference (ICGA95)*, pages 318–325, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[19] I. Page. Constructing hardware-software systems from a single description. *Journal of VLSI Signal Processing*, 1(12):87–107, Jan. 1996. Kluwer Academic Publishers.

[20] W. Press, B. Flannery, S. Teukolsky, and W. Vetterling. *Numerical recipes, the art of scientific computing*. Cambridge University Press, 1986.

[21] B. Shackleford, G. Snider, R. Carter, E. Okushi, M. Yasuda, K. Seo, and H. Yasuura. A high performance, pipelined, FPGA-based genetic algorithm machine. *Genetic Programming and Evolvable Machines*, 2(1):33–60, Mar. 2001.

[22] D. Stiliadis and A. Varma. FAST: An FPGA-based simulation testbed for ATM networks. *Proc. ICC'96*, 1996.

[23] M. Tommiska and J. Vuori. Hardware implementation of GA. In J. Alander, editor, *Proceedings of the Second Nordic Workshop on Genetic Algorithms and their Applications (2NWGA)*, Vaasa, Finland, 1996.

[24] S. Wolfram. Random sequence generation in cellular automata. *Adv. Appl. Math.*, 7:123–169, 1986.

[25] Xilinx. Pseudo random number generator. www.xilinx.com/xcell/xl35/xl35_44.pdf, Dec. 2001.

Appendix A

Results of the Diehard Tests

This appendix contains the results of running the Diehard tests for all RNGs in this paper. Max score represents the case where an RNG fails all the tests.

Table 3: Diehard test results for all RNGs considered in this paper.

Test	Max score	LFSR	EQG	32LFSR	IDCA	32CA	True	Mother
Birthday	36	36	8	2	0	8	0	0
Overlapping permutation	8	8	0	4	8	8	0	0
Binary Rank 32x32	8	8	2	8	2	6	0	0
Binary Rank 6x	104	104	40	8	140	70	4	6
Bitstream	80	80	0	0	80	80	4	0
Overlapping pairs tests	328	328	188	94	328	320	6	2
Count the ones (stream)	8	8	8	8	8	8	0	0
Count the ones (specific)	100	100	42	30	100	100	2	4
Parking Lot	44	4	0	0	4	2	0	0
Minimum Distance	4	4	0	4	4	4	0	0
3D spheres	84	4	0	2	4	2	4	4
Squeeze	4	4	0	0	4	4	0	0
Overlapping Sums	44	44	0	0	6	0	2	2
Runs	16	16	0	2	16	8	0	2
Craps	8	8	0	0	8	12	0	0
Total	876	756	288	162	676	640	22	20

Crossover Operators for a Hardware Implementation of GP using FPGAs and Handel-C

Peter Martin
Department of Computer Science,
Essex University, Wivenhoe Park,
Colchester, Essex, UK
petemartin@ntlworld.com

Riccardo Poli
Department of Computer Science,
Essex University, Wivenhoe Park,
Colchester, Essex, UK
rpoli@essex.ac.uk

Abstract

This paper analyses the behavior of the crossover operator in a hardware implementation of Genetic Programming using Field Programmable Gate Arrays. Three different crossover operators that limit the lengths of programs are analysed: A truncating operator, a limiting operator that constrains the lengths of both offspring and a limiting operator that only constrains the length of one offspring. The latter has some interesting properties that suggest a new method of limiting code growth in the presence of fitness.

1 Introduction

Previous work has described an implementation of Genetic Programming using a Field Programmable Gate Array (FPGA) and a high level language to hardware compilation system called Handel-C [6]. This was tested using the XOR and symbolic regression problems. Further work described a pipelined implementation that improved the performance and demonstrated that the technique could be used to solve the artificial ant problem [7]. In both cases the work concentrated on the implementation issues and increasing the clock speed of the implementation, but put to one side the study of the behavior of the system. Now that the raw throughput issues have been considered it is time to look at the behavior, and investigate and analyse some alternative implementation issues.

Because of limited hardware resources in an FPGA and to keep the design simple and therefore efficient, the maximum program size is fixed. To ensure that crossover always generates programs that are shorter than the maximum length, the crossover operator limits the program size by truncating programs that exceed the maximum length. The effect of this decision is investigated in this paper and some other alternative methods of limiting program length are explored.

The paper begins with a brief description of the implementation of a GP system using FPGAs. This is followed by an analysis of the crossover operator, with comparisons to standard tree based GP [3]. We then consider two alternative crossover operators and analyse their behavior. The analysis is then discussed and finally some further work is suggested and some conclusions are given.

2 A Hardware Implementation of GP using FPGAs

Implementing GP in hardware is motivated by the potential speedups that can be obtained. The platform chosen for this work is a Field Programmable Gate Array (FPGA). An FPGA is a reconfigurable device than can be programmed to perform a wide range of logic functions. A typical FPGA is arranged as an array of configurable logic cells, input-output circuits and programmable interconnections, and is shown in Figure 1.

Traditionally FPGAs have been programmed using hardware design languages such as VHDL[1], but alternative approaches using high level language to hardware compilation techniques have also been developed, in which a high level imperative language is used to generate the configuration information for the FPGA. Handel-C [1] is one example of this technology, and has been used for the work described in this paper.

For a detailed review of previous work using FPGAs in Evolutionary Computing refer to [6].

2.1 Target Hardware

The target hardware is a Celoxica RC1000 FPGA development board fitted with a Xilinx XCV2000E Virtex-E

[1] VHDL is a standard hardware design language. It stands for VHSIC Hardware Design Language. VHSIC itself stands for Very High Speed Integrated Circuit.

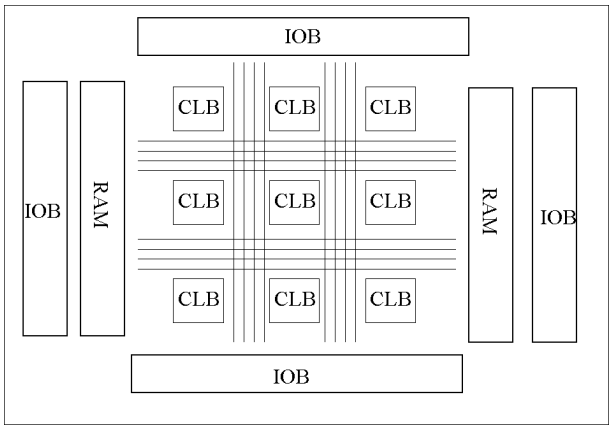


Figure 1: Typical FPGA architecture. The CLBs are the configurable logic blocks, IOBs are the Input Output Blocks and the RAMs are on-chip Random Access memory blocks.

FPGA having 43,200 logic cells and 655,360 bits of block ram. The board also has a PCI bridge that communicates between the RC1000 board and the host computer's PCI bus, and four banks of Static Random Access Memory (SRAM). Fast switches isolate the FPGA from the SRAM, allowing both the host CPU and the FPGA to access the SRAM, though not concurrently.

2.2 Program Representation

Handel-C does not support a stack, which means that a standard tree based representation is not straightforward to implement because recursion is not supported by the language. An alternative to a tree representation is a linear representation which has been used by others to solve some hard GP problems, for example [8]. Using a linear representation, a program consists of a sequence of words which are interpreted by the problem specific fitness function. The hardware design uses a linear program representation with a fixed maximum size. Choosing a fixed maximum size made the storage of programs in on-chip RAM and off-chip RAM efficient and simple to implement. Consequently a method of limiting the program size during crossover was needed. The first implementation used a truncating crossover. This is compared to a second method of limiting lengths, called the limiting crossover operator.

3 Analysis of the crossover operator

Two separate implementations were used for the analysis. Firstly, a simple program that simulated the effects of GP crossover was used to show the expected program length distributions in the absence of fitness. We refer to this as the GP simulator in this paper. Secondly, the hardware implementation was used to obtain results both with and without fitness. The test problem for all the experiments where fitness is used is the artificial ant problem.

3.1 Artificial Ant

This popular test problem was originally described by Jefferson [2] and in the context of GP by Koza [3]. It involves finding a program for an ant-like machine that enables it to navigate its way round a trail of food on a 32x32 toroidal grid of cells within a fixed number of time steps. In the hardware implementation the function set differs from the standard example in only having two functions: $\mathcal{F} = \{IF\_FOOD, PROGN2\}$ where IF_FOOD is a two argument function that looks at the cell ahead and if it contains food it evaluates the first terminal, otherwise it evaluates the second terminal. $PROGN2$ evaluates its first and second terminals in sequence. The terminal set $\mathcal{T} = \{LEFT, RIGHT, MOVE, NOP\}$, where $LEFT$ and $RIGHT$ change the direction the ant is facing, $MOVE$ moves the ant one space forwards to a new cell, and if the new cell contains food, the food is eaten. NOP is a no-operation terminal and has no effect on the ant but is included to make the number of terminals a power of 2, which simplifies the hardware logic. Each time $LEFT$, $RIGHT$ or $MOVE$ is executed, the ant consumes one time step. The run stops when either all the time steps have been used, or the ant has eaten all the food. This test problem was chosen because it is known to be a hard problem for GP to solve [5].

All the results use the Santa Fe trail, which has 89 pellets of food. Each experiment was run 500 times and the mean of all the runs taken. Unless stated otherwise, the population size is 1024, the maximum program length is 31 and all experiments were run for 31 generations. The ant was allocated 600 timesteps. The probability of selecting crossover was 67%, mutation 10% and straight reproduction 23%.

3.2 Behavior Analysis

The measurement of overall GP behavior is frequently limited to plotting the mean population fitness vs. generation. This is shown for the artificial ant problem using the hardware implementation in Figure 2 over 500 runs. This will be used as a baseline when looking at changes to the original design. However, when looking for the reasons to explain why a feature of an operator or representation has an effect, raw performance gives us a very restricted view of what is happening, and more analytical methods are needed. One such method is to consider one or more aspects of the internal population dynamics during a run. Recently a lot of work has been done to develop exact schema theories for Genetic Programming [10][11], which, among other things, give us a description of the expected changes

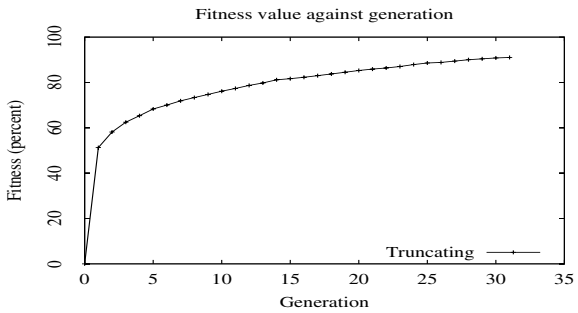


Figure 2: GP Performance of the artificial ant problem using a hardware GP system. Average of 500 runs.

in the program length distribution during a GP run. The asymptotic distribution of program lengths is important to us because it is a way of comparing the sampling behavior (search bias) of different crossover operators and replacement strategies.

Starting with the GP simulator with a uniform initial length distribution and ignoring the effects of fitness, Figure 3 shows the expected length distribution for generations 0,1,10 and 31. In this case there is no maximum program size. This agrees with the results in [11] where the distribution asymptotically converges to a discrete Gamma distribution.

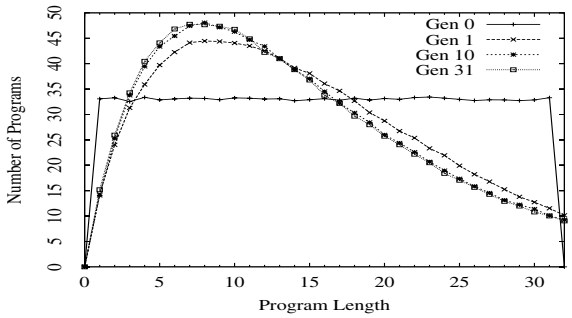


Figure 3: Program length distribution for standard GP crossover using a linear program representation, a global replacement strategy, non-steady state without fitness.

3.3 Truncating Crossover Operator

This crossover operator ensures programs do not exceed the maximum program length by selecting crossover points in two individuals at random and exchanging the tail portions up to the maximum program length. Crossovers that result in programs exceeding the maximum length are truncated at the maximum length. This crossover operator was devised to minimize the amount of logic required and the number of clock cycles needed. This is illustrated in Figure 4. For two programs a and b that have lengths l_a and

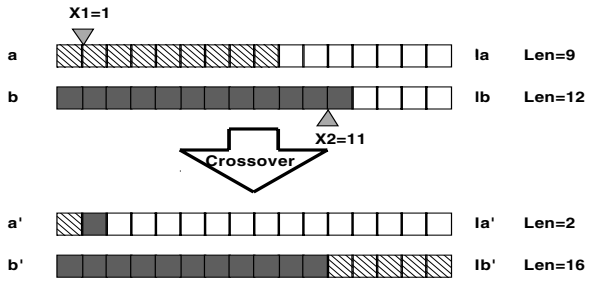


Figure 4: Truncating crossover operator

l_b, two crossover points x_a and x_b are chosen at random so that $0 \leq x_a < l_a$ and $0 \leq x_b < l_b$. The program size limit is L_{max}. After crossover the new lengths are $l'_a = \min((x_a + l_b - x_b), L_{max})$ and $l'_b = \min((x_b + l_a - x_a), L_{max})$.

When the GP simulator is modified to implement the truncating crossover, the result is shown in Figure 5 without fitness. The behavior of the hardware implementation using

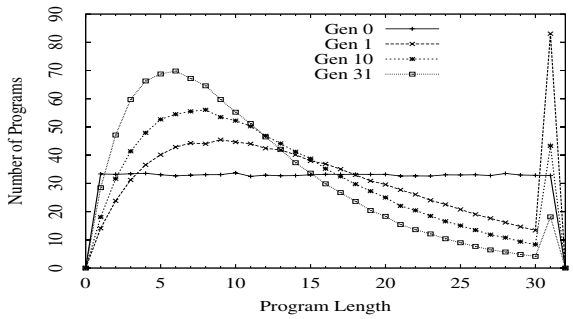


Figure 5: Program length distribution with truncating crossover for standard GP without fitness.

the truncating crossover operator is shown in Figure 6. A feature of these results is that there is initially a large peak at the maximum program size of 31, but in subsequent generations the distribution tends to resemble a Gamma distribution like the one in Figure 3. However, it is important to note that it is not the same Gamma distribution, because the mean program length tends to decrease with this crossover operator. The reason is that with the truncation the amount of genetic material removed from the parents when creating the offspring may be bigger than the amount of genetic material replacing it. The differences between Figures 5 and 6 are believed to arise because the simulator uses generational GP, while the hardware implementation uses steady state GP.

When fitness is used, the length distribution changes as shown in Figure 7, but it still retains some of the features of a Gamma distribution. The striking feature is the large

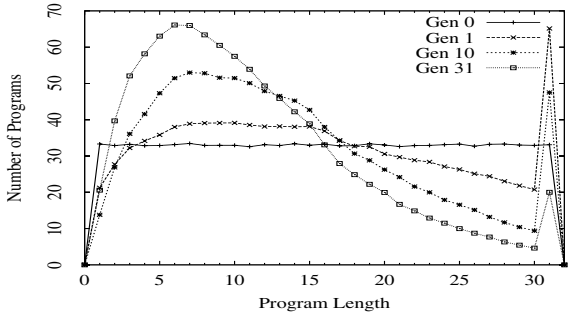


Figure 6: Program length distribution using truncating crossover using a linear program representation without fitness. From the hardware implementation.

peak at the maximum program length limit which represents nearly 10% of the total population.

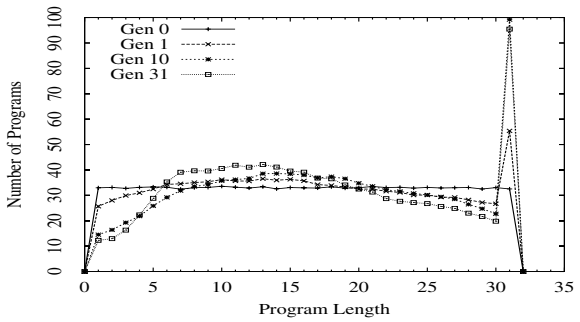


Figure 7: Program length distribution using truncating crossover using a linear program representation with fitness. From the hardware implementation.

3.4 Limiting Crossover Operator

An alternative method of ensuring that programs do not exceed the fixed limit is to repeatedly choose crossover points until both programs are below the program size limit L_{max}. For two programs a and b, with lengths l_a and l_b, two crossover points x_a and x_b are chosen so that $0 \leq x_a < l_a$ and $0 \leq x_b < l_b$. After crossover the new lengths are simply $l'_a = x_a + l_b - x_b$ and $l'_b = x_b + l_a - x_a$. If $l'_a > L_{max}$ or $l'_b > L_{max}$ the selection of x_a and x_b is repeated until $l'_a \leq L_{max}$ AND $l'_b \leq L_{max}$.

This is the approach taken in lilgp (versions 1.02 and 1.1) when the `keep_trying` parameter is enabled [12] to limit the tree depth and the total number of nodes in a program tree during crossover. When this crossover operator is implemented in the GP simulator the program length distribution changes, as shown in Figure 8. A feature of this result is that the mean program length moves towards smaller values. After 31 generations, the population size distribution

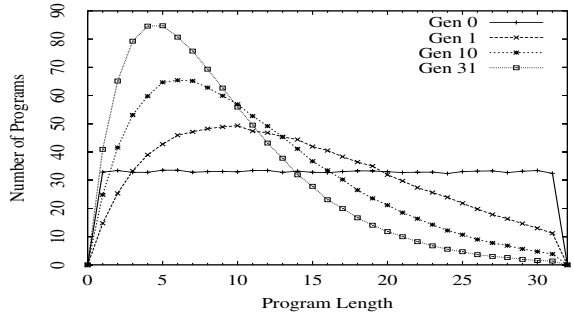


Figure 8: Program length distribution using limiting crossover operator and a global replacement strategy without fitness.

shape resembles the one produced with standard GP.

When this method of limiting the program length was implemented in the hardware version, we obtained the distribution shown in Figure 9. In contrast to the GP simulator the program length distribution remains reasonably static between generations 1 and 31. In an effort to understand

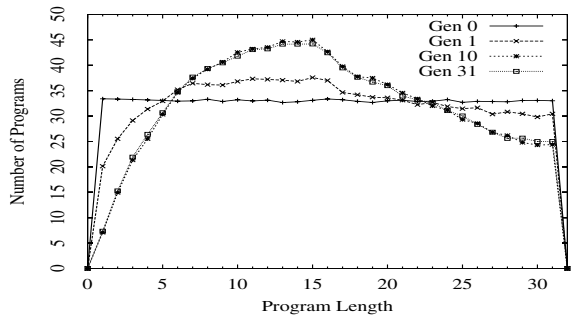


Figure 9: Program length distribution using limiting crossover without fitness, from the hardware implementation.

the different behavior between the results in Figures 8 and 9 it was noted that the hardware implementation required both of the offspring programs a' AND b' to be shorter than L_{max} but that the simulation only considered one offspring at a time, effectively requiring a' OR b' to be shorter. The latter case is referred to as the single-child variant in the rest of this paper, and the original the dual-child variant. In the case of the single-child variant, if one of the programs was larger than the maximum, it was simply discarded and the parent substituted in its place, and if both children were larger than the limit, the two crossover points would be chosen again. If both children were smaller than the limit, they would both be available as candidates in the next generation. When the hardware implementation was modified to incorporate the single-child variant limiting method, the result shown in Figure 10 was obtained, closely matching

that from the simulation. Again, the difference between Figure 8 and Figure 10 is believed to be due to the use of steady-state GP in the hardware implementation. When fitness is enabled using the dual-child variant, there is a large bias in favor of longer programs as shown in Figure 11. An interesting artifact of this graph is the sharp rise in program lengths for generations 10 and 31 above length 15. This is likely to be due to the distribution of fitness in the program search space and can be seen as a form of what is commonly termed bloat. However, when the program length distribution using the single-child variant was plotted, shown in Figure 12, the length distribution peaks at around the mean of L_{max}. This unexpected behavior is interesting since it appears to have avoided the phenomenon of bloat.

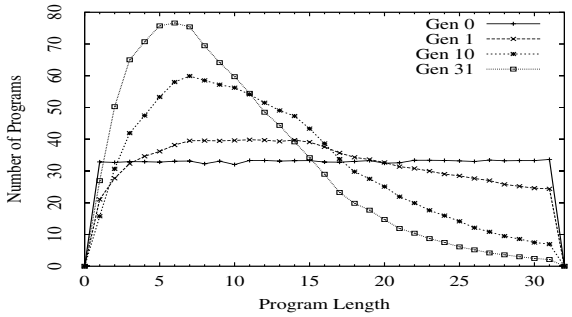


Figure 10: Program length distribution using limiting crossover without fitness and the single-child variant. From the hardware implementation.

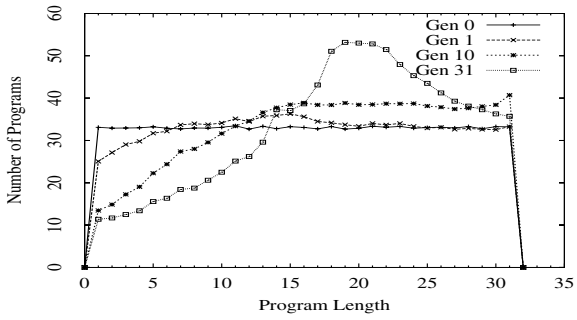


Figure 11: Program length distribution using limiting crossover with fitness and the dual-child variant. From the hardware implementation.

The effect of using the limiting crossover operator with and without the single-child variant on the behavior of the system is shown in Figure 13 together with the original behavior. This graph shows that all three crossover implementations have a similar rate of improvement, with the limiting crossover operator with single-child variant maybe performing slightly better on the ant problem.

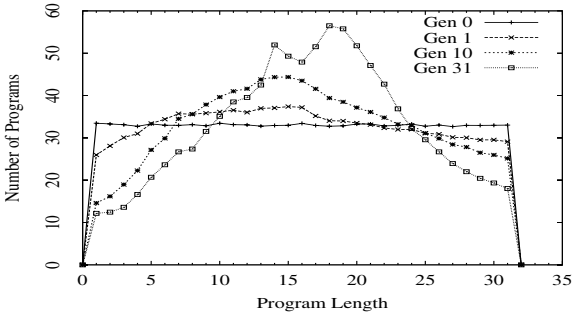


Figure 12: Program length distribution using limiting crossover with fitness and the single-child variant. From the hardware implementation.

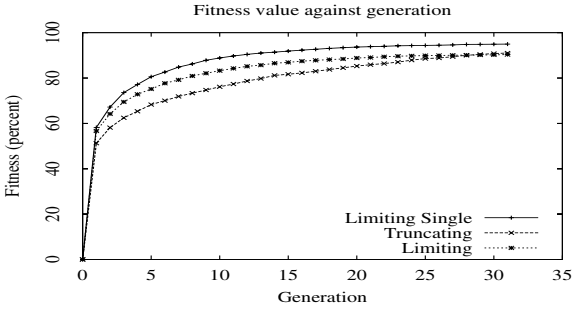


Figure 13: Comparative GP behavior of the hardware implementation for the ant problem using truncating crossover and limiting crossover.

Finally, the distribution of 100% correct program lengths was measured for truncating and both limiting crossovers. The hardware implementation was run 500 times, and if a 100% correct program was generated, the length was recorded. These are shown in Figures 14, 15 and 16 respectively.

From these plots we can see that truncating crossover has allowed GP to find more 100% correct programs than the limiting crossover using the dual-child variant. However, when using the single-child variant, limiting crossover found the most 100% correct programs.

It is interesting to note that the results shown in Figure 13 do not obviously show this difference in the outcome, highlighting the weakness of using the standard measure of performance.

The results shown in Figures 14,15 and 16 suggest that for the artificial ant problem implemented in hardware, programs of length 4 or 5 are most likely to be correct. It was then observed that the peak program length in Figure 12 was larger than length 4. From this it was conjectured that if the maximum program length was reduced from 32, moving the peak closer to the program length that occurred

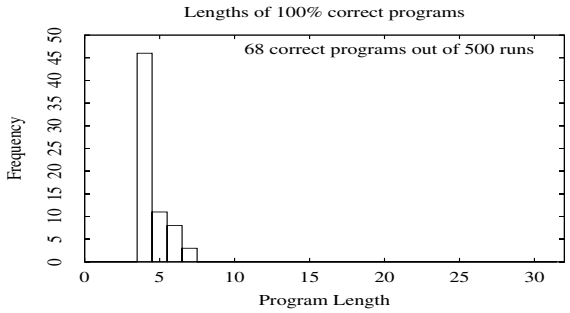


Figure 14: Distribution of lengths of 100% correct programs using the truncating crossover operator.

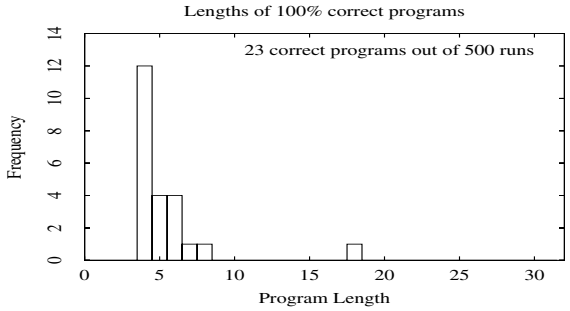


Figure 15: Distribution of lengths of 100% correct programs using the dual-child variant limiting crossover operator.

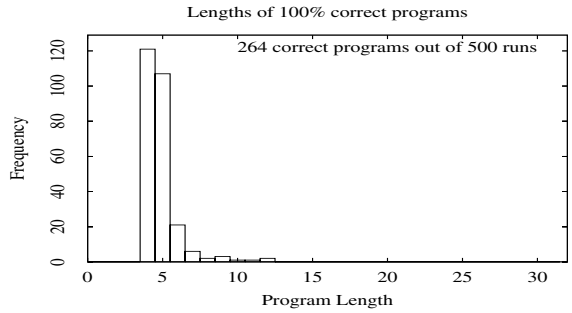


Figure 16: Distribution of lengths of 100% correct programs using the the single-child variant limiting crossover operator.

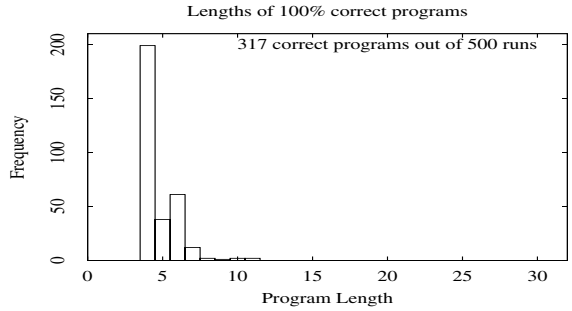


Figure 17: Distribution of lengths of 100% correct programs using the the single-child variant limiting crossover operator and a length limit of 16

most frequently, that GP may find even more successful programs. Two further experiments were therefore performed using maximum lengths of 16 and 8. The results of running the hardware implementation with these modified lengths is shown in Figures 17 and 18.

This confirmed the idea that, by limiting the program lengths that GP is allowed to create, that GP produced more 100% correct programs. The corresponding program length distributions are shown in Figures 19 and 20. These both have similar characteristics to Figure 12 and show that the program length distribution peaks close to the peak of the successful programs.

4 Discussion

The differences between the dual-child and single-child variants can be explained by considering first the dual-child case. Starting with a uniform distribution of program lengths $0 < l \leq L_{max}$, the average program length is given by $L_{avg} = \frac{L_{max}}{2}$ and the average crossover point is $\frac{L_{avg}}{2}$. Every crossover produces two offspring, the average length of which is $\frac{L_{max}}{2}$, with one smaller and one larger program produced. When one of the offspring exceeds L_{max} both crossover points are re-selected until both programs satisfy the length constraint. The result is that the average program length using this crossover will remain $\frac{L_{max}}{2}$. However, in the single-child case, only one child needs to meet the length constraint. With one long and one short offspring, the short offspring will be more likely to satisfy the constraint and so be selected for propagation. Because the shorter program is preferred, the mean program length will tend to continually decrease. In summary, in the absence of fitness, the single-child variant selects programs that are on average smaller than $\frac{L_{max}}{2}$. In the presence of fitness we believe that this pressure to decrease the mean program length competes with the well documented tendency of GP programs to grow in the presence of fitness. The result is that when using the single length constraint and an upper bound on the program length, the program length distribution does not have a strong bias to longer lengths.

A side effect of using the single child variant is that when a long program is rejected, a copy of the parent is propagated to the next generation. This means that even if crossover is used as the only operator, a proportion of straightforward reproduction will be present.

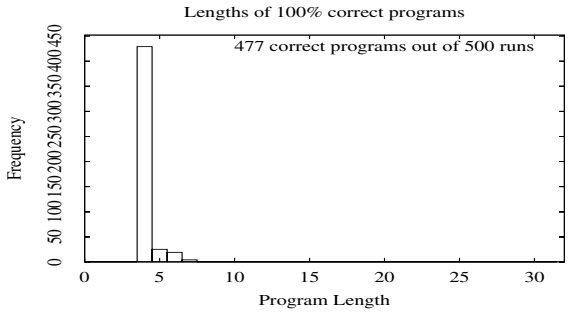


Figure 18: Distribution of lengths of 100% correct programs using the the single-child variant limiting crossover operator and a length limit of 8

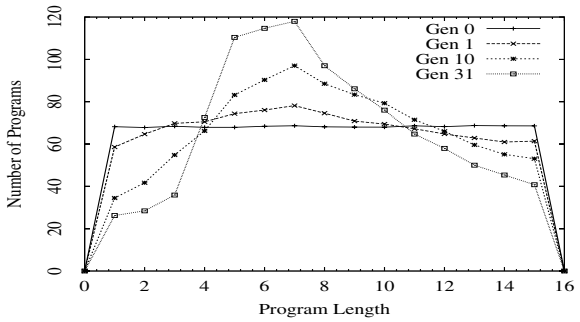


Figure 19: Program length distribution using limiting crossover with fitness and the single-child variant. Maximum length limited to 16. From the hardware implementation.

A practical penalty of the limiting crossover approach is that multiple passes may be required to obtain two crossover points that satisfy the length constraints. Depending on the implementation this could have an impact on the time needed to complete a GP run. In practice for most problems the time required for crossover in a standard GP system is much smaller than the time for evaluating programs, and so will only extend the time required by a small factor. In the hardware implementation, crossover is performed in parallel with evaluation, so there will be no impact for most problems where fitness evaluation takes longer than selection and breeding. For the artificial ant problem implemented in hardware, the limiting crossover operators did not have any effect on the overall performance of the design, both the clock speed and number of clock cycles remained the same as the truncating crossover implementation. It is worth noting that the single-child limiting crossover will need fewer iterations to find a legal offspring, so this will have a smaller effect on the overall performance.

The effect of adjusting the program length limit so that the peak in the length distribution is closer to the peak of optimal program lengths suggests that allowing programs to be unlimited in length may be detrimental to using GP effectively.

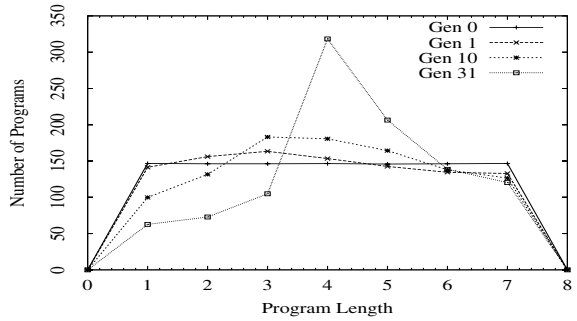


Figure 20: Program length distribution using limiting crossover with fitness and the single-child variant. Maximum length limited to 8. From the hardware implementation.

5 Further work

From the results in [10] we would expect similar behavior when these techniques are applied to standard tree based GP, and this is currently being investigated.

Other techniques have been suggested for controlling the program size during evolution, such as the smooth operators [9], homologous and size fair operators [4] which could also be adapted to a hardware implementation.

So far, only one problem has been analysed using the hardware implementation of GP and to get a more complete picture of the effects of the design decisions more problems need to be implemented and analysed.

6 Conclusions

This analysis, based on measuring the program length distributions was prompted by the results from the work on a general schema theory of GP. It has led us to an implementation of crossover that allows us to constrain the maximum program lengths. For the ant problem implemented in hardware we have discovered a mechanism that avoids the effects of unconstrained program growth, and indeed allows us to obtain more correct programs.

In conclusion, all three crossover operators are effective in the hardware implementation when applied to the artificial ant problem, with the single-child limiting crossover performing ahead of the other two. The behavior of the single-child limiting crossover in the presence of fitness is interesting and suggests another mechanism for controlling code growth.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments. The first author would like to thank Marconi plc, Celoxica Ltd. and Xilinx Inc. for supporting this work.

References

[1] Celoxica. Web site of Celoxica Ltd. www.celoxica.com, 2001. Vendors of Handel-C. Last visited 15/June/2001.

[2] D. Jefferson, R. Collins, C. Cooper, M. Dyer, M. Flowers, R. Karf, C. Taylor, and A. Wang. Evolution as a theme in artificial life: The Genesys/Tracker system. In C. Langton, editor, *Artificial Life II*. Addison-Wesley Publishing Company Inc., 1992.

[3] J. Koza. *Genetic programming: on the programming of computers by means of natural selection*. MIT Press, Cambridge, MA, USA, 1992.

[4] W. Langdon. Size fair and homologous tree genetic programming crossovers. In W. Banzhaf, J. Daida, A. Eiben, M. Garzon, V. Honavar, M. Jakiela, and R. Smith, editors, *Proceedings of the genetic and evolutionary computation conference*, volume 2, pages 1092–1097, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[5] W. Langdon and R. Poli. Why ants are hard. In J. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. Fogel, M. Garzon, D. Goldberg, H. Iba, and R. Riolo, editors, *Genetic programming 1998: proceedings of the third annual conference*, pages 193–201, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[6] P. Martin. A Hardware Implementation of a Genetic Programming System using FPGAs and Handel-C. *Genetic Programming and Evolvable Machines*, 2(4):317–343, 2001.

[7] P. Martin. A pipelined hardware implementation of genetic programming using FPGAs and Handel-C. In *Eurogp2002*, 2002.

[8] P. Nordin and W. Banzhaf. Evolving turing-complete programs for a register machine with self-modifying code. In L. Eshelman, editor, *Genetic algorithms: proceedings of the sixth international conference (ICGA95)*, pages 318–325, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[9] J. Page, R. Poli, and W. Langdon. Smooth uniform crossover with smooth point mutation in genetic programming: A preliminary study. In R. Poli, P. Nordin, W. Langdon, and T. Fogarty, editors, *Genetic programming, proceedings of EuroGP'99*, volume 1598 of *LNCS*, pages 39–49, Goteborg, Sweden, 26-27 May 1999. Springer-Verlag.

[10] R. Poli. General schema theory for genetic programming with subtree-swapping crossover. In J. Miller, M. Tomassini, P. Lanz, C. Ryan, G. Andrea, B. Tettamanzi, and W. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 143–159, Lake Como, Italy, Apr. 2001. EvoNET, Springer-Verlag.

[11] R. Poli and N. F. McPhee. Exact schema theorems for GP with one-point and standard crossover operating on linear structures and their application to the study of the evolution of size. In J. F. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 126–142, Lake Como, Italy, 18-20 Apr. 2001. Springer-Verlag.

[12] D. Zongker and B. Punch. Lilgp 1.01 user's manual. Technical report, Michigan State University, USA, 26 Mar. 1996.

Using schema theory to explore interactions of multiple operators

Nicholas Freitag McPhee
Division of Science and Mathematics
University of Minnesota, Morris, USA
mcphee@mrs.umn.edu
320-589-6300

Riccardo Poli
Department of Computer Science
University of Essex, UK
rpoli@essex.ac.uk
+44-1206-872338

Abstract

In the last two years the schema theory for Genetic Programming (GP) has been applied to the problem of understanding the length biases of a variety of crossover and mutation operators on variable length linear structures. In these initial papers, operators were studied in isolation. In practice, however, they are typically used in various combinations, and in this paper we present the first schema theory analysis of the complex interactions of multiple operators. In particular, we apply the schema theory to the use of standard subtree crossover, full mutation, and grow mutation (in varying proportions) to variable length linear structures in the one-then-zeros problem. We then show how the results can be used to guide choices about the relative proportion of these operators in order to achieve certain structural goals during a run.

1 Introduction

Most (if not all) Genetic Programming (GP) operators have a variety of biases with respect to both the syntax and the semantics of the trees they produce. These biases can work against or in favor of the biases implied by the fitness function, which makes understanding these biases crucial to understanding the behavior of and relationships among the various operators.

These interactions can be quite complex, however, and consequently understanding them can be difficult. While there is a considerable literature examining the interactions of mutation and crossover in areas like Genetic Algorithms (GAs), there is much less reported work on the interactions of operators in GP. Notable exceptions include the work of O'Reilly [10], Banzhaf, *et al* [1], and Luke and Spector [5, 6, 4]. These studies are primarily experimental in nature, and all suggest that understanding operator interactions is difficult. It would thus be useful to have a theoretical approach to these problems that might allow us to better understand operator interactions, and choose combinations of operators in a more principled manner.

In the last few years work on schema theory for GP has made huge progress, generating not only an exact theory, but also one applicable to a variety of operators used in practice, including: one-point crossover [12, 14, 11, 13], standard and other subtree-swapping crossovers [14, 16, 7], different types of subtree mutation and headless chicken crossover [15, 8], and the class of homologous crossovers [17].

In [16, 7] we showed how these recent developments in GP schema theory can be used to better understand the biases induced by the standard subtree crossover when genetic programming is applied to variable length linear structures. In particular we showed that subtree crossover has a very strong bias towards oversampling shorter strings and, in some senses, works against bloat. In [15, 8] we derived exact schema equations for subtree mutation on linear structures, using both the full and grow methods to generate the new, random subtrees. Iterating those equations on both a flat fitness landscape and a needle-in-a-haystack style problem, called the one-then-zeros problem, we showed that both of these subtree mutation operators have strong biases with regard to the population's length distribution. Similar to the bias of subtree crossover, we found that these mutation operators are strongly biased in favor of shorter strings in both these fitness domains.

In this paper we combine the schema theory for different operators and apply them to the problem of better understanding the behavior produced by their interaction. Studying these complex interactions is particularly easy using the schema formalization because we can simply use a weighted sum of the schema equations generated for each operator in isolation. We also show how the theory can be used to design competent GP systems by guiding the choice

of combinations of operators together with their parameter settings.

The work reported here is all on GP with linear structures (not unlike those used in, e.g., [9, 2]), although the schema theory on which it is based is much more general. We have chosen in these applications to focus on linear structures because the theoretical analysis is more manageable and the computations are more tractable. This has yielded a number of important results for the linear case, and preliminary results further suggest that many of the key ideas here are also applicable (at least in broad terms) to the non-linear tree structures typically used in GP.

In Sec. 2 we will introduce the schema theorem for GP using linear structures, standard crossover and mutation, and we will show how easily the theory for different operators can be combined. We then apply the theory in Sec. 3 to the one-then-zeros problem and use the theory to both predict and better understand the changes in the distribution of fit individuals and of sizes (Sec. 4). We finish with some conclusions and ideas for future research (Sec. 5).

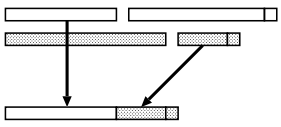
2 Schema theory for GP on linear structures

2.1 Operators

In this paper we will consider three common GP operators: the standard subtree-swapping GP crossover operator, and the full and grow mutation operators. Each operator acts by removing a non-empty suffix of an individual and replacing it with a new suffix, with the production of that suffix being the primary difference between the operators.

More formally, in a linear-structure GP where $\mathcal{F}$ is the set of non-terminal nodes and $\mathcal{T}$ is the set of terminal nodes, individuals can be seen as sequences of symbols $c_0 c_1 \ldots c_{N-1}$ where $c_i \in \mathcal{F}$ for $i < N-1$ and $c_{N-1} \in \mathcal{T}$. Each of the operators, then, starts by removing a non-empty suffix $c_j c_{j+1} \ldots c_{N-1}$ (where j is chosen uniformly such that $0 \leq j < N$) and replacing it with a new non-empty string.[1]

In the case of crossover, the new string is taken to be a suffix $d_{j'} d_{j'+1} \ldots d_{N'-1}$ of another parent $d_0 d_1 \ldots d_{N'-1}$, where j' (which could differ from j) is chosen uniformly such that $0 \leq j' < N'$.



[1]The requirement that suffixes be non-empty while prefixes are allowed to be empty comes from standard practice in GP. It does, however, create a number of mild but annoying asymmetries which often clutter up the analysis (see, e.g., [18]).

Both full and grow mutation generate the new suffix randomly, and they differ in how the new random subsequences are generated, and in particular how their sizes are determined. In full mutation, the subsequence has a specified length D; thus non-terminals are selected uniformly from $\mathcal{F}$ until length $D-1$ is reached, at which point a terminal is selected uniformly from $\mathcal{T}$. In grow mutation, on the other hand, one chooses from the set of *all* functions and terminals every time, only terminating the creation of the subsequence when a terminal is chosen; thus for grow mutation there is no *a priori* limit on the size of the resulting sequences.

2.2 Schema theory definitions

In this section we will present a series of crucial definitions that allow us to represent schemata, and count and build instances of schemata.

Just as we defined a linear GP structure to be a sequence of symbols, we will also define a linear GP *schema* as the same kind of sequence $c_0 c_1 \ldots c_{N-1}$ except that a new "don't care" symbol '=' is added to both $\mathcal{F}$ and $\mathcal{T}$.[2] Thus schemata represent sets of linear structures, where the positions labelled '=' can be filled in by any element of $\mathcal{F}$ (or $\mathcal{T}$ if it is the terminal position). A few examples of schema are:[3]

- $(=)^N$: The set of all sequences of length N.
- $1(=)^a$: The set of all sequences of length $a+1$ starting with a 1.
- $1(0)^a$: The singleton set containing the symbol 1 followed by a 0's.

Now that we can represent schemata, we present a series of definitions that allow us to count instances of schemata.

Definition 1 (Proportion in population) *$\phi(H,t)$ is the proportion of strings in the population at time t matching schema H. For finite populations of size M, $\phi(H,t) = m(H,t)/M$, where $m(H,t)$ is the number of instances of H at time t.*

Definition 2 (Selection probability) *$p(H,t)$ is the probability of selecting an instance of schema H from the population at time t. This is typically a function of $\phi(H,t)$, the fitness distribution in the population, and the details*

[2]This new '=' symbol plays a role similar to that of the '#' "don't care" symbol in GA schema theory. For historical reasons, however, '#' has been assigned another meaning in the more general version of the GP schema theory [14].

[3]We will use the superscript notation from theory of computation, where x^n indicates a sequence of n x's.

of the selection operators. With fitness proportionate selection, for example, $p(H,t) = \phi(H,t) \times f(H,t)/\overline{f}(t)$*, where* $f(H,t)$ *is the average fitness of all the instances of* H *in the population at time* t *and* $\overline{f}(t)$ *is the average fitness in the population at time* t*.*

Definition 3 (Transmission probability) $\alpha(H,t)$ *is the probability that an instance of the schema* H *will be constructed in the process of creating a new individual for the population at time* $t+1$ *out of the population at time* t*. This will typically be a function of* $p(K,t)$*, the various schemata* K *that could play a role in constructing* H*, and the details of the various recombination and mutation operators being used.*

Definition 4 (Creation probability) $\pi_{mut}(H,t)$ *is the probability that some GP subtree mutation operator will generate a new, random subtree that is an element of the schema* H *in generation* t*.*

To clarify which operator we are working with, we introduce specialized forms of the transmission probability function α, namely α_{xo} for the transmission probability due specifically to crossover, α_{FULL} for the transmission probability due specifically to subtree mutation using the full method, and α_{GROW} for the transmission probability due specifically to subtree mutation using the grow method.

We can now model the standard evolutionary algorithm as the transformation

$$\phi(H,t) \stackrel{\text{select}}{\longrightarrow} p(H,t) \stackrel{\substack{\text{mutate}\\ \text{XO}}}{\longrightarrow} \alpha(H,t) \stackrel{\text{sample}}{\longrightarrow} \phi(H,t+1).$$

Here the arrows indicate that some new distribution (on the RHS of the arrow) is generated by applying the specified operation(s) to the previous distribution (on the LHS). So, for example, the process of selection can be seen as a transformation from the distribution of schemata $\phi(H,t)$ to the selection probability $p(H,t)$. A crucial observation is that, for an *infinite* population, $\phi(H,t+1) = \alpha(H,t)$ for $t \geq 0$, which means we can iterate these transformations to *exactly* model the behavior of an infinite population over time.

To formalize the creation of instances of a linear schema $H = c_0 c_1 \ldots c_{N-1}$ we define

$$\begin{aligned} u(H,i,k) &= c_0 c_1 \ldots c_{i-1} (=)^{k-i} \\ l(H,i,n) &= (=)^{n-N+i} c_i c_{i+1} \ldots c_{N-1} \end{aligned}$$

Here $u(H,i,k)$ is the schema of length k matching the leftmost i symbols of H, and $l(H,i,n)$ is the schema of length n matching the rightmost $N-i$ symbols of H.[4] The important property of u and l is that if one uses standard crossover to crossover *any* instance of $u(H,i,k)$ at position i with *any* instance of $l(H,i,n)$ at position $n-N+i$, the result will be an instance of H, provided[5] $k+n > N$, and $0 \uparrow (N-n) \leq i < N \downarrow k$. Further, these are the *only* ways to use standard crossover to construct instances of H, so these definitions fully characterize the mechanism for constructing instances of H.

2.3 The schema theorem

[7, 8] provide schema theorems for each of our three operators when used in isolation. Here we extend these results to the case where all three operators can be used in the same run, each with specified proportions. Since we use exactly one operator to generate any given individual, the probability that we construct an instance of a schema (i.e., $\alpha(H,t)$) is simply the sum of the probabilities of each specific operator constructing such an instance, each weighted by the likelihood of choosing that operator. This leads to the following:

Theorem 1 (Schema theorem for combined operators) *For GP on linear structures using standard crossover with probability* p_{xo}*, full mutation with length* D *and probability* p_{FULL}*, and grow mutation with probability* p_{GROW}*, such that* $p_{xo} + p_{FULL} + p_{GROW} = 1$*, we have*

$$\begin{aligned} \alpha(H,t) &= p_{xo} \times \alpha_{xo}(H,t) \\ &+ p_{FULL} \times \alpha_{FULL}(H,t) + p_{GROW} \times \alpha_{GROW}(H,t) \end{aligned}$$

where

$$\begin{aligned} \alpha_{xo}(H,t) &= \sum_{\substack{k>0 \\ n>0 \\ k+n>N}} \Bigg(\frac{1}{k \times n} \\ &\quad \times \sum_{0 \uparrow (N-n) \leq i < N \downarrow k} p(u(H,i,k),t) \\ &\qquad \times p(l(H,i,n),t) \Bigg), \\ \alpha_{FULL}(H,t) &= \sum_{\substack{k>0 \\ 0 \leq i < N \downarrow k}} \Bigg(\frac{1}{k} \times p(u(H,i,k),t) \\ &\qquad \times \pi_{FULL}(c_i c_{i+1} \ldots c_{N-1}) \Bigg) \\ \alpha_{GROW}(H,t) &= \sum_{\substack{k>0 \\ 0 \leq i < N \downarrow k}} \Bigg(\frac{1}{k} \times p(u(H,i,k),t) \\ &\qquad \times \pi_{GROW}(c_i c_{i+1} \ldots c_{N-1}) \Bigg), \end{aligned}$$

and $q = |\mathcal{F}|/(|\mathcal{F}| + |\mathcal{T}|)$.

[4] u and l are based on operators U and L (see, e.g., [14]) which match the *upper* and *lower* parts of general, non-linear, GP schemata.

[5] We will use $\uparrow$ as a binary infix *max* operator, and $\downarrow$ as a binary infix *min* operator.

Due to space restrictions we simply report the general expressions for the quantities α_{xo}, α_{FULL}, and α_{GROW} for the linear case without providing any proofs. The interested reader can find these, together with extensive characterizations of the behavior of crossover and mutation when used separately, in [7, 8].

3 The one-then-zeros problem

We will now apply the Schema Theorem to the *one-then-zeros problem*. We will start by defining and motivating the problem, and then show how the schema theorem can be used to better understand the effects of multiple operator interaction on this problem.

3.1 One-then-zeros problem definition

In this problem we have $\mathcal{F} = \{0, 1\}$ and $\mathcal{T} = \{0\}$, where both 0 and 1 are unary operators. This gives us a problem that is essentially equivalent to studying variable length strings of 0's and 1's, with the constraint that the strings always end in a 0. Fitness in this problem will be 1 if the string starts with a 1 and has zeros elsewhere, i.e., the string has the form $1(0)^a$ where $a > 0$; fitness will be 0 otherwise.

One of the reasons for studying this problem is that under selection and crossover this problem induces bloat [7], whereas this does not happen when using the full and grow mutation operators [8]. The key advantage of this problem is that in order to fully and exactly describe the length-evolution dynamics and the changes in solution frequency of infinite populations, it is necessary to keep track of only two classes of schemata: those of the form $(=)^N$ and those of the form $1(0)^a$. Unfortunately most problems are not so restricted, and one is typically forced to track the proportion of many (possibly intractably many) more schemata.

3.2 Analyzing one-then-zeros

To apply the schema theorem to the one-then-zeros problem one needs to calculate the probabilities α_{xo}, α_{FULL}, and α_{GROW} for both of the schema $(=)^N$ and $1(0)^a$, and the probabilities π_{FULL} and π_{GROW} for both of the schema $1(0)^a$ and $(0)^a$. These can be calculated from the equations reported above and are also provided in explicit form in [7, 8], so we will not re-derive these results here.

If we assume an infinite population, we can numerically iterate the equations in the Schema Theorem to better understand the behavior of an infinite GP population on this problem. Tracking these distributions over time becomes expensive in terms of computational effort.[6] A crucial point, though, is that these equations only need to be run once, and have no stochastic effects. They are *exact* calculations of the relevant quantities (up to the limitations of the floating point representation), and once computed need never be computed again. This is in contrast to typical empirical results in evolutionary computation, where combinations of large populations and multiple runs are necessary to smooth out the stochastic effects, and even then there is no guarantee that any two sets of runs will have similar behavior.

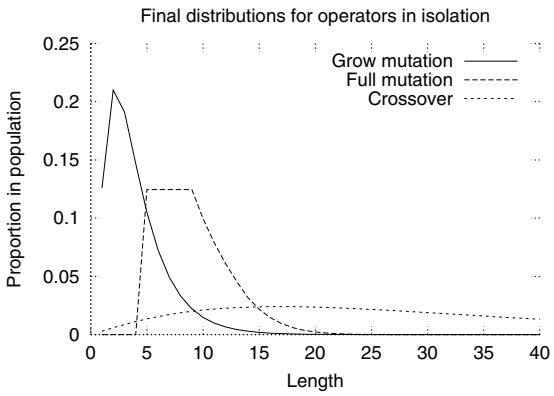


Figure 1: The distributions of lengths after 50 generations when using the three recombination operators individually on the one-then-zeros problem. The tail of the crossover distribution continues past the right hand side of the graph, with lengths above 300 still having proportions above 10^{-10}.

4 One-then-zeros results

We know (see, e.g., [7, 16, 8, 15]) that each of these operators has significant biases when used on its own, and Fig. 1 summarizes some of the earlier results by presenting the final length distributions for each of the operators when acting alone on the one-then-zeros problem. This makes it clear that the three operators all have very different length biases, which suggests that they may indeed demonstrate interesting behaviors when used in combination.

We can now iterate this new combined schema equation to study these combined interactions and their biases, and to use such results to guide the choices of the proportions of operators to help satisfy a variety of goals. As an example in this paper we will consider the following goals:

1. Avoid both bloating and shrinking, by having the average size after 50 generations be as close as possible to the initial average size.

[6] We have found, though, that ignoring values of α below some small threshold (we have used 10^{-10}) seems to have little impact on the numeric results and can greatly speed up the calculations since it significantly slows the growth of the number of strings that need to be tracked.

2. Avoid both bloating and shrinking (as above), but also maximize the number of correct individuals.
3. Maximize the proportion of small solutions (as opposed to just short strings).
4. Reach a state where the proportion of $1(0)^{30}$ exceeds 0.01 as early as possible.

In all these simulations we will be applying the three operators discussed earlier (standard subtree crossover, full mutation, and grow mutation) on the one-then-zeros problem. A depth limit $D = 5$ will be used for full mutation. Our initial population will consist of equal proportions (10% each) of the strings $1(0)^i$ for $1 \leq i \leq 10$; thus the average length in the initial population is 6.5.

To study the interaction of the operators, the schema equations from Theorem 1 were iterated 66 different times, using each of the legal combinations of proportions of (standard) crossover, grow mutation, and full mutation with values from the set $\{0, 0.1, 0.2, 0.3, \ldots, 0.9, 1\}$. We'll use triples of the form $(p_{xo}, p_{\text{FULL}}, p_{\text{GROW}})$ to indicate a combination of parameter settings where the first is always the proportion of crossover, the second the proportion of full mutation, and the third the proportion of grow mutation.

4.1 General observations

While the majority of these iterations had converged after 50 generations, there were several which had not. These were typically those with sufficiently high crossover probabilities that bloat was occurring and the average lengths were clearly still growing after 50 generations. As an example, the configuration (0.8, 0, 0.2) has an average length of 7.98 after 50 generations, and is thus not a terrible solution to the problem of avoiding bloat and shrinkage as defined in Sec. 4.2 below. It seems highly likely, however, that if we were to continue iterating the equations with these parameters for another 100 generations we would get higher average length, thereby doing a worse job of meeting the goal of avoiding bloat and shrinkage. This isn't necessarily a concern, however, since actual GP runs always have a finite number of generations. Thus if we know we're likely to run our GP for 100 generations, we can iterate these schema equations and try to find settings that meet our goals (whatever they happen to be) at the end of 100 generations *regardless of whether further generations would take us away from our goals*.

It should also be noted that the initial uniform distribution of lengths is very unstable in the sense that any combination of operators will generate a very different distribution immediately in the first generation. As an example, the settings (0.1, 0.7, 0.2) have an average length after 50 generations that's very close to the average length in the initial distribution (6.5). The distribution itself (shown in Fig. 3) is far from uniform, however. This seems to be a general property of "interesting" operators, namely that they have a favored length distribution that they move to quite quickly, and while fitness can modify that tendency, it rarely eliminates it entirely.

4.2 Avoid bloat and shrinkage

In our first example the goal will be to avoid both bloating and shrinkage by searching for a collection of operator probabilities such that the average length after 50 generations is as close as possible to the initial average size.

Out of our 66 configurations, five had a final average fitness that was less than 0.15 away from the initial average of 6.5 (see Table 1); the next closest combination of parameter settings had an absolute difference of over 0.23. Note that in each case the proportion of grow mutation was 0.2. In fact the 20 configurations whose final average lengths were closest to 6.5 all had small non-zero proportions for grow mutation (between 0.1 and 0.4); at the same time, however, those 20 configurations had a broad range of full mutation proportions (ranging from 0 to 0.9) and crossover proportions (from 0 to 0.8). Those combinations where the proportion of crossover was over 0.5, however, all had average lengths that were still climbing after 50 generations, so it's likely that they would continue to diverge from 6.5 if we iterated the equations for more generations. Thus the crucial factors for long-term size stability *in this problem* seem to be a small non-zero proportion of grow, and a crossover proportion of at most 0.5 so the sizes don't bloat above 6.5.

Most (but not all) of the configurations where the proportion of grow was 0.2 had final average lengths close to 6.5; the smallest average length after 50 generations was 6.36 (for $(0.4, 0.4, 0.2)$), and the largest was 7.98 (for $(0.8, 0, 0.2)$). As discussed above, however, those parameter sets with higher crossover proportions probably hadn't converged after just 50 generations, and their final averages would likely continue to grow if we iterated more generations. Taking that into account the range of final average lengths is quite small, being from 6.35 to 6.46 when the proportion of grow is 0.2 and the proportion of crossover is at most 0.5.

Looking at Fig. 2, we can see that in each of these cases there was an initial jump away from 6.5 (caused by the instability of the initial uniform length distribution), followed by a fairly rapid convergence to an average value close to 6.5. The slowest to converge was the case where we had 50% crossover, and that curve in fact looks similar to the bloating seen in [7], with an asymptote close to 6.5.

Fig. 3 shows the final distribution of lengths for each of these five parameter settings. While each of these distribu-

XO	Full	Grow	Diff. from 6.5	Prop. fit
0.1	0.7	0.2	0.081	0.19
0.2	0.6	0.2	-0.045	0.26
0.3	0.5	0.2	-0.131	0.31
0.4	0.4	0.2	-0.147	0.37
0.5	0.3	0.2	-0.045	0.43

Table 1: Parameter settings for the five configurations that came closest to having the same average length after 50 generations as the average length of the initial distribution. "XO" is the proportion of crossover, and "Full" and "Grow" are the proportions of full and grow mutation. "Diff. from 6.5" is the difference between the actual final average length for this set of parameters and the initial average length (6.5); negative values mean that the final average length was less than 6.5. "Prop. of fit" is the proportion of the individuals produced in the last generation that were fit.

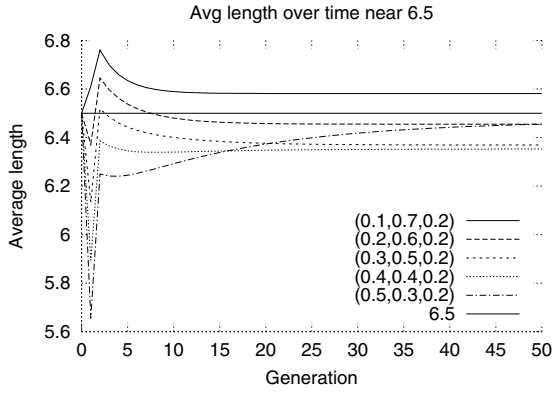


Figure 2: Average lengths over time for the five collections of parameter settings leading to a final average length closest to the initial average length (6.5).

tions has an average length that is nearly equal to that of the initial uniform distributions, none of these distributions is remotely uniform. They instead exhibit combinations of features seen in earlier studies of using single recombination operators on this problem (see Fig. 1). In each case, for example, we see a peak at length=5 which is due to full mutation with depth 5, and the height of the peak is clearly correlated to the full mutation probability.

4.3 Avoid bloat and shrinkage, maximizing correct proportion

In the preceding example we looked for parameter settings that avoided both bloat and shrinkage. It's possible, however, that this goal was met at the expense of correctness. A given collection of parameter settings could, for example, generate the desired average size, but have a very low proportion of fit individuals. This would in turn greatly reduce the effective population size since most of the generated individuals can never be selected for recombination.

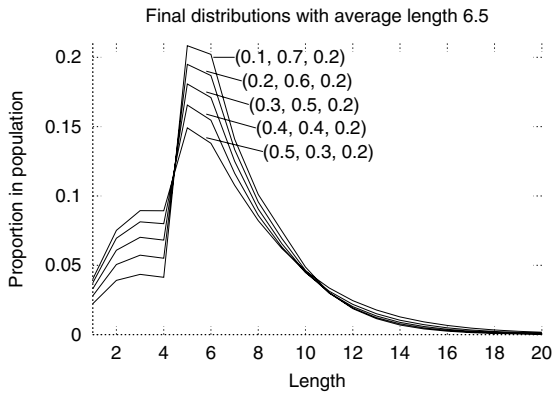


Figure 3: Final length distributions of the five parameter settings whose final average length was closest to the initial average length (6.5).

We can assess this by looking at the proportion of correct individuals in the final generation for each set of parameter values, and we indeed see that there are substantial differences among these five configurations (see Table 1), with the values ranging from 0.19 to 0.43. It's also clear that increased probabilities of crossover correspond with increased proportion of fit individuals. This is not surprising since increased probabilities of crossover also correspond to decreased probabilities of full mutation, and full mutation is rarely going to produce a fit offspring in this problem [8]. Crossover on the other hand, has a high probability of generating correct offspring, especially when given two correct, fairly long individuals as parents [7].

Another approach to optimizing these two criteria would be to start by identifying the configurations with high proportion of fit individuals in the final generation, and then choosing from those the parameter settings that also lead to final average lengths near 6.5. The settings with the highest proportion of fit individuals are those with high crossover probabilities, but these configurations also have the highest final average lengths (because high crossover probabilities lead to bloat in this problem [7]). One of the best settings is (0.8, 0.0, 0.2), which has a final average length of 7.98 (nearly 1.5 nodes longer than the original average of 6.5) but a final proportion of fit individuals of 0.65 (about 0.22 higher than the proportion generated by (0.5, 0.3, 0.2)).

4.4 Maximize proportion of small solutions

Now consider the case where we want to minimize the average size of the *fit* individuals (i.e., those of the form $1(0)^a$ for $a > 0$). There are quite a few combinations of parameter settings that lead to average size of fit individuals that are just above 3. The three smallest are (0, 0, 1), (0.1, 0, 0.9), and (0.2, 0, 0.8) with final average sizes of fit individu-

als 3.02, 3.04, and 3.07 respectively. This suggests that (for this problem) the best way to make short correct solutions is to primarily use grow mutation, with small amounts of crossover being acceptable as well. Adding small amounts of full mutation doesn't lead to much bigger average sizes (the average size for (0, 0.1, 0.9), for example, is 3.13) despite the fact that depth mutation always generates strings of length at least $1 + D$ (or 6 in our case). This is probably due to the fact that full mutation will very rarely generate correct individuals in this problem.

If one wanted to further maximize the proportion of fit individuals, then the three candidate combinations have progressively increasing proportion of fit individuals; the highest is (0.2, 0, 0.8), which has a proportion of 0.41. If we look more broadly, we find that (0.6, 0, 0.4) has a somewhat higher proportion of fit individuals after 50 generations (0.49), with the average size of fit individuals being only slightly higher (3.54).

4.5 Find solutions of length 31 quickly

For our final example we will consider the goal of finding solutions of the form $1(0)^{30}$ as quickly as possible. There are a variety of motivations for this sort of goal, but one might be that instead of only having a two level fitness function, we might have a three level fitness function: Fitness 0 for individuals that don't have the "one then zeros" pattern, fitness 1 for individuals of the form $1(0)^a$, $a \neq 30$, and fitness 2 for individuals of the form $1(0)^{30}$. If we further assume that our run will terminate as soon as we discover a target individual $1(0)^{30}$, then dynamics of such a run are identical to the original one-then-zeros problem, except they terminate upon discovery of a target individual.

Thus we can use our schema theory results to discover what parameter settings lead most quickly to the discovery of a target individual. Because of the infinite population assumption, however, we may find that early in a run there is a *very* small, but still positive, proportion of target individuals, yet with such small proportions the likelihood is minuscule of actually generating a target individual that quickly in a "real" (finite population) run. We will, therefore, look for the collection of parameter settings that first achieves a proportion of target individuals exceeding 0.01.

Only four of our tested parameter settings ever obtain a proportion of at least 0.01 target individuals (see Table 2), with all crossover (1, 0, 0) reaching the target the most quickly (in 26 generations). Adding small amounts of full mutation still allows the goal to be satisfied, but even a proportion of 0.3 is enough to increase the number of generations by 12. Grow mutation clearly interferes with this goal, as none of the four parameter settings that achieve the goal have any grow mutation.

XO	Full	Grow	First gen to 0.01
1.0	0.0	0.0	26
0.9	0.1	0.0	28
0.8	0.2	0.0	32
0.7	0.3	0.0	38

Table 2: Parameter settings for the four configurations that eventually achieve a proportion of 0.01 target individuals $1(0)^{30}$. The first three columns are as in Table 1. "First gen to 0.01" is the first generation for a given collection of parameter settings where the proportion of target individuals exceeded 0.01.

If we relax the target proportion to 0.001 there are a total of 12 parameter settings that achieve this new goal. Of these only four have non-zero grow mutation probabilities, all of which are the lowest possible value (0.1). Similarly, all but three of these 12 settings have crossover probabilities exceeding 0.5, although one (0.3, 0.7, 0) managed to reach the target of 0.001 in 21 generations despite the low crossover probability. It's interesting to note, however, that two of the settings with non-zero grow mutation probabilities ((0.7, 0.2, 0.1) and (0.6, 0.3, 0.1)) both reached the goal more slowly (in 23 and 29 generations respectively) despite having much high crossover probabilities.

It's not terribly surprising that crossover is useful in increasing the length of fit strings, since we've previously seen that crossover can lead to bloat (presumably due to replication accuracy) [7]. Further, one would expect both mutation operators to at least slow down the process of generating a target string containing thirty 0's, and thus having length 31 (see [18] for details):

- Given a parent string of length l, full mutation generates (on average) a string of length roughly $l/2+D$, so full mutation tends to generate shorter offspring once $l > 2D$. Since $D = 5$ in our examples, full mutation will tend to reduce the size of strings once their lengths begin to exceed 10.
- Given a parent string of length l, grow mutation in the one-then-zeros problems will generate (on average) a string of length roughly $l/2 + 3$. Thus grow mutation will tend to reduce the size of strings once their lengths begin to exceed 6.

What's perhaps more surprising is that grow mutation interferes with the process of finding a target string so much more than full mutation does. The likely reason is that grow mutation is more likely to produce fit offspring than full mutation (see [8, 18] for details). Because of the infinite population assumption, generating unfit offspring has no substantial effect on the dynamics of the system, as doing so has no effect on the selection probabilities. Generating short, fit individuals, however, will change the dy-

namics by increasing the probability that short individuals are selected as parents in the next generation. In our case its likely that grow mutation creates a sufficient number of short, fit strings that it can significantly hamper the process of generating fit strings of length 31.

5 Conclusions and future work

It's clear, then, that there is a fairly complex set of interactions between these three recombination operators, making it quite difficult to guess *a priori* what proportions of operations would aid in satisfying goals that might be important in a particular domain. For this problem, however, we were able to iterate the schema equations on many different combinations of operator proportions, generating a useful map of the interactions.

In this paper the number of different combinations of proportions was small enough to make manual searches for desirable values feasible. With more operators, or a larger variety of different proportions, the number of combinations would quickly grow out of control, making it prohibitive to iterate the equations for every combination and then search the results by hand. Since this is essentially just another parameter optimization problem, one possibility would be to apply a GA, although in many cases something simpler like a hill-climber would probably also work. Another possibility (which could potentially dramatically reduce the number of different combinations that would need to be iterated) would be to use factorial design of experiments [3].

Perhaps the key observation here is that there is clearly no "best" set of operator proportions, and that the desirability of a combination of operators will depend critically on the specific goals. It is therefore particularly important that we have tools that help us understand not only the general interactions of operators, but also understand the more specific interactions in order to guide our choices.

Acknowledgments

Both authors would like the thank the Dagstuhl International Conference and Research Center for Computer Science, and the organizer of Dagstuhl Seminar No. 02031, for providing the wonderful environment that made this paper possible.

References

[1] W. Banzhaf, F. D. Francone, and P. Nordin. The effect of extensive use of the mutation operator on generalization in genetic programming using sparse data sets. In H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature IV, Proceedings of the International Conference on Evolutionary Computation*, volume 1141 of *LNCS*, pages 300–309, Berlin, Germany, 22-26 Sept. 1996. Springer Verlag.

[2] D. S. Burke, K. A. D. Jong, J. J. Grefenstette, C. L. Ramsey, and A. S. Wu. Putting more genetics into genetic algorithms. *Evolutionary Computation*, 6(4):387–410, Winter 1998.

[3] R. Feldt and P. Nordin. Using factorial experiments to evaluate the effect of genetic programming parameters. In R. Poli, W. Banzhaf, W. B. Langdon, J. F. Miller, P. Nordin, and T. C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 271–282, Edinburgh, 15-16 Apr. 2000. Springer-Verlag.

[4] S. Luke. *Issues in Scaling Genetic Programming: Breeding Strategies, Tree Generation, and Code Bloat*. PhD thesis, Department of Computer Science, University of Maryland, A. V. Williams Building, University of Maryland, College Park, MD 20742 USA, 2000.

[5] S. Luke and L. Spector. A comparison of crossover and mutation in genetic programming. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 240–248, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[6] S. Luke and L. Spector. A revised comparison of crossover and mutation in genetic programming. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 208–213, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[7] N. F. McPhee and R. Poli. A schema theory analysis of the evolution of size in genetic programming with linear representations. In *Genetic Programming, Proceedings of EuroGP 2001*, LNCS, Milan, 18-20 Apr. 2001. Springer-Verlag.

[8] N. F. McPhee, R. Poli, and J. E. Rowe. A schema theory analysis of mutation size biases in genetic programming with linear representations. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC 2001*, Seoul, Korea, May 2001.

[9] M. O'Neill and C. Ryan. Grammatical evolution. *IEEE Transaction on Evolutionary Computation*, 5(4), 2001.

[10] U.-M. O'Reilly. *An Analysis of Genetic Programming*. PhD thesis, Carleton University, Ottawa-Carleton Institute for Computer Science, Ottawa, Ontario, Canada, 22 Sept. 1995.

[11] R. Poli. Exact schema theorem and effective fitness for GP with one-point crossover. In D. Whitley, D. Goldberg, E. Cantu-Paz, L. Spector, I. Parmee, and H.-G. Beyer, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 469–476, Las Vegas, July 2000. Morgan Kaufmann.

[12] R. Poli. Hyperschema theory for GP with one-point crossover, building blocks, and some new results in GA theory. In R. Poli, W. Banzhaf, and *et al.*, editors, *Genetic Programming, Proceedings of EuroGP 2000*. Springer-Verlag, 15-16 Apr. 2000.

[13] R. Poli. Exact schema theory for genetic programming and variable-length genetic algorithms with one-point crossover. *Genetic Programming and Evolvable Machines*, 2(2), 2001.

[14] R. Poli. General schema theory for genetic programming with subtree-swapping crossover. In *Genetic Programming, Proceedings of EuroGP 2001*, LNCS, Milan, 18-20 Apr. 2001. Springer-Verlag.

[15] R. Poli and N. F. McPhee. Exact GP schema theory for headless chicken crossover and subtree mutation. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC 2001*, Seoul, Korea, May 2001.

[16] R. Poli and N. F. McPhee. Exact schema theorems for GP with one-point and standard crossover operating on linear structures and their application to the study of the evolution of size. In *Genetic Programming, Proceedings of EuroGP 2001*, LNCS, Milan, 18-20 Apr. 2001. Springer-Verlag.

[17] R. Poli and N. F. McPhee. Exact schema theory for GP and variable-length GAs with homologous crossover. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

[18] J. E. Rowe and N. F. McPhee. The effects of crossover and mutation operators on variable length linear structures. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

Evolving Compression preprocessors with genetic programming

Johan Parent
Vrije Universiteit Brussel
Pleinlaan 2
Brussels 1050, Belgium
parentjo@vub.ac.be

Ann Nowe
Vrije Universiteit Brussel
Pleinlaan 2
Brussels 1050, Belgium
asnowe@info.vub.ac.be

Abstract

We present a new approach for applying genetic programming to lossless data compression. Unlike programmatic compression the evolved programs are preprocessors. These preprocessors aim at enhancing the compression rate of the given data by transforming it. The entropy based fitness function is both fast and independent of the type of information being processed. The obtained results are encouraging in sense that significant improvements can be achieved. Furthermore the required computation time is much smaller than in the case of programmatic compression, making the presented approach more viable. We used a strongly typed GP kernel. The kernel offers the extra advantage of being able to exploit parallel execution through the island model.

1 Introduction

Compression has been a research topic for many years. In 1940 Claude Shannon already studied what later became information theory. His research has determined the theoretical limits of data compression. Current research focuses mainly on the development of application specific compression algorithms. Generic lossless compression algorithms can however be considered at a stand still. A recent algorithm is the Burrows-Wheeler transform (1993) [1] which is used in the bzip2 compression program.

Record compression rates can be achieved using a good model of the data, e.g. true color images. Here we investigate how a program can be evolved to transform the data so that it matches the model used by a given compression algorithm. The exact transformation is not explicitly known but one can formulate certain conditions the latter should possess (see section 3).

Our attention goes toward lossless compression algorithms. Examples of popular lossless compression programs are gzip [8] and Winzip [2].

This document is structured as follows. In section 2 related research is presented. The problem and the chosen approach are described in section 3. Section 4 details the experimental setup. Sections 5 and 6 present the results and the conclusion respectively.

2 Related work

Evolutionary algorithms have been used in the past for data compression purposes. Two approaches can be distinguished.

Genetic algorithms were used to find parameters for a compression algorithm in order to maximize compression Driesen [4]. Feiel and Ramakrishnan [6] have used genetic algorithms to optimize the compression of color images using vector quantization.

Genetic programming was used for what is called *programmatic compression*. This approach is closely related to algorithmic complexity were one looks for the shortest program that produces the given data [13]. De Falco et al. [5] have used genetic programming for string compression. Fukunaga and Stechert [7] have used genetic programming for lossless compression of gray-scale images. Nordin and Banzhaf [10] achieved lossy programmatic compression of images and sound. Noteworthy is the fact that [10] [7] both used a genocompiler for their experiments. This software eliminates the function call overhead incurred by other systems during the evaluation of the individuals. Luke [9] reports on a relative improvement in speed of 2000 times compared to LISP code and of 100 times compared to interpreted C (like used for this experiment).

[1]

3 Preprocessing

Data compression is already a highly specialized domain. Therefore it seems too far fetched to use genetic programming to generate an algorithm that would compress data and do so in a competitive way.

We formulate our objective as follows: instead of aiming for a program that recodes the data we seek a transformation.

This transformation is applied to the data before compressing it. It is used as a preprocessing step in the entire compression process. Of course, after decompressing the data the transformation needs to be reversed in order to obtain the original data (since we focus on lossless compression).

Figure 1: The preprocessing takes place before the actual compression and is reverserd after decompression.

Such a preprocessing program can be formalized as a function P that works on a string S over an alphabet. The length of a string S is denoted as $|S|$. C represents a data compression algorithm. The result we are looking for is a transformation P, so that the condition denoted in equation 1 holds.

$$|C(P(S))| < |C(S)| \tag{1}$$

Stating the problem in these terms makes it easy to fold it into the genetic programming framework since both the program we are looking for, being P, and the fitness function are easily identified. Formulating the problem in this way has some serious disadvantages though. First, computing the result of equation1 is rather expensive. The compression algorithm has to be applied to the transformed data for every individual in the population. Second, the transformation will depend on the compression algorithm used. To avoid this problem we reformulated it using a metric from information theory, the entropy.

[1] Notwithstanding the increase in computation speed [7] [10] report runs lasting several tens of hours on powerful workstations. The results presented here required far less time while working on a bigger amount of data.

3.1 Entropy

Consider a message as a series of symbols. The entropy can be thought of as a measure for the *information content* of a message. The entropy gives the average information content of a symbol [2], this is typically expressed in bits per symbols. The formula for the entropy is given below, note that P_i represents the probability of symbol i in the message.[3]

$$H = -\sum_{\mathrm{n}} P_i . \log P_i \tag{2}$$

Using the entropy (equation 2) we have a means to determine how much information is present in a given message. The entropy is the theoretical lower bound on the size of the data (in bits). Any representation of the data with a size lower than the one *predicted* by the entropy loses information. Important is the fact that this measure is independent of the type of data being represented.

3.2 Reducing the entropy

We will evolve a transformation for the given data using the entropy as an objective criterion. The purpose of this transformation is: lowering the entropy of a message (data). The information content can be reduced without loss by exploiting redundancies that might be hidden in the data (as will be shown in section 5.2). By lowering the entropy we reduce the information content, this means that the data can be recoded to occupy less *space*. This property is independent of any compression algorithm. Compression algorithms are designed so as to recode data in order to match the *real* size of the data.

The instruction set of the genetic programming software is designed to reduce the entropy, albeit under the good circumstances. It is up to the evolutionary pressure to bring forth the best transformation. Using the entropy we now can define a new condition for the transformation we wish to evolve.

$$\frac{H_{out} \times L_{out}}{H_{in} \times L_{in}} < 1 \tag{3}$$

Equation 3 is a computationally cheaper fitness function. It is furthermore independent of any data com-

[2] Entropy has a much more rigorous mathematical foundation but the description given here suffices for the purpose of this text.

[3] The model used here is a first order model (marginal probability). Higher order models are based on conditional probabilities.

pression algorithm. Since on the average a symbol represents H bits of information, a message with length L gives $H \times L$ bits of information in total. The formula expresses that the total information of the transformed data has to be lower than the information of the initial data. Note that we impose no limit on the length of the transformed data. Since we do not immediately compress the data, L_{out} can either be greater than or equal to L_{in}.

4 Experimental setup

The setup used for the experiments will now be presented. The transformation one seeks is represented by an S-expression. The function and terminal set used here relies on a simple virtual machine. The use of a virtual machine gives a limited function and terminal set with clear semantics without sacrificing performance or introducing limits on the data size.

The instruction set of the virtual machine has been *wrapped* to form the function and terminal set used by the genetic programming software. To structure the S-expressions strong typing has been used.

4.1 Input data

For the experiments the Canterbury Corpus [3] as well as various bitmaps and word processor files were used. This means that the size of the input tape usually exceeded the order of several kilobytes and grew even up to more than 1 megabyte.

4.2 Parallel and strongly typed

The approach presented here uses a strongly typed genetic programming kernel written in C that produces LISP-like programs. This package is a modified version of the Lil-gp package [12]. This modified version can run on parallel using multi processor machines and clusters of workstations [11].

4.3 Virtual machine

The virtual machine bears some resemblance with an automaton as it uses an input and an output tape. The instruction set can be divided into two categories. Instructions that control the input tape and instructions that process the data read from it. Note that there are no operations that directly influence the output tape in the instruction set. The output tape is manipulated implicitly whenever new data is read from the input tape. This is done through two buffers inside the machine as depicted in the figure 2. The processed data, which is stored in the output buffer, is copied to the output tape when new data is loaded in the input buffer.

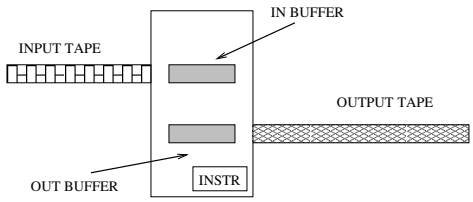


Figure 2: A simple virtual machine has been defined which bears some resemblance with an automaton. The internal buffers are resized to accommodate the data read from the tape.

A status variable is used in order to control the correct operation of the virtual machine. In certain circumstances the execution of an instruction can be illegal. In this case the status of the virtual machine is set accordingly. Whenever an attempt has been made to perform an illegal instruction the execution of subsequent instructions is aborted. The execution of the program is thereby limited to the correct portion of the program.

4.3.1 Tape operations

The machine needs to read data from the input tape. To that end two instructions were defined. Note that the size of the internal buffers is adjusted so that it can contain all the symbols read from tape.

- load : this instruction takes 1 parameter, an integer. This instruction reads the specified number of symbols from the input tape and copies them in the input buffer. This operation is always valid.
- fwd : is identical to load except that it does not take any parameter. Instead it uses the value of the parameter passed to the last *load* instruction. If there was no load instruction preceding the fwd this operation is invalid.

A third rewind instruction makes it possible to apply several operations on data read from the input tape.

- rew : no new data is read or written from or to the tapes. Instead the previously read **and** processed data is processed again. The content of the output buffer is simply copied in the input buffer.

4.3.2 Data operations

Next to operations for reading data, operations were defined for processing the data. The data operations

are performed on the data present in the input buffer. Data produced by data operation is stored in the output buffer.

Since we are working with lossless compression each of the data operations needs to be reversible. Each of these operations needs to be reversible in order to be useful within the lossless compression context. Hence, for each data operation there is a reverse operation defined in the virtual machine. These operations are however not made available in the instruction set of the genetic programming software.

The choice of the data operations was guided by their capability to reduce the entropy of the data. This is typical done by exploiting patterns that exist in the data. A stride parameter was introduced for some operations. The stride specifies the step size when going over the data. For example all the symbols 3 positions apart (in this case the stride would be 3). The stride allows data operations to work on patterns that may be spread within the data.

- dpcm : stands for differential pulse code modulation. $\{x_0, x_1, x_2, x_3, \ldots, x_n\} \Rightarrow \{x_0, x_1 - x_0, x_2 - x_1, x_3 - x_2, \ldots, x_n - x_{n-1}\}$ This operation has a stride parameter.

- min : This operation uses the average of all the symbols (binary representation) in the input buffer. A symbol is replaced by the difference between the average and the symbol. This operation does expand the data by 1 symbol, the symbol which represents the average[4].

  $\{x_0, x_1, x_2, x_3, \ldots, x_n\} \Rightarrow \{x_{avg}, x_0 - x_{avg}, x_1 - x_{avg}, x_2 - x_{avg}, x_3 - x_{avg}, \ldots, x_n - x_{avg}\}$ This operation has a stride parameter.

- raw : no transformation is applied at all to the data. $\{x_0, x_1, x_2, x_3, \ldots, x_n\} \Rightarrow \{x_0, x_1, x_2, x_3, \ldots, x_n\}$ This may seem like a quite useless instruction. But it allows for *jumps* in the data processing since we maybe do not need to transform the entire tape. An alternative would be the definition of a *jump* tape instruction.

- pec : pseudo exponential code. This operation produces an output with double size of the input. The output data represents an input symbol as a couple of numbers. The output is based on the number of the input symbol. A couple is a *pseudo* exponent and a *pseudo* remainder. The pseudo exponent is the largest exponent used as a power of 2 which doesn't exceed the input number, e.g. 3 for 10. The remainder is the difference between the power of two and the input number, in the previous example this would be 2 (10 - 2^3). The reason for calling this *pseudo*[5] is that it has been modified for numbers bigger than 128.

- mtf : move to front. This operation is a standard encoding scheme which uses a map of all the possible symbols. The symbols are replaced by their position in the map. Each time a symbol is replaced by its current position in the map the latter is updated. The symbol is put in front of the map thereby assigning it a small number. This allows to encode the symbols that appeared recently with small numbers.

- inv : inversion. $\{x_0, x_1, x_2, x_3, \ldots, x_n\} \Rightarrow \{MAX(x) - x_0, MAX(x) - x_1, MAX(x) - x_2, MAX(x) - x_3, \ldots, MAX(x) - x_n\}$ Here $MAX(x)$ represents the maximum value that can be represent by the data type used to represent the symbols. This operation has a stride parameter.

- sub : substitution. This operation substitutes the symbol that appears most frequently in the input buffer with the most frequent symbol in the output tape so far. This operation has a stride parameter.

4.4 Types

Typing is used in the first place to structure the programs that can be evolved. Evaluating the functions and terminals corresponds to the execution of an instruction by the virtual machine. For the experiments four types were used for the functions and terminals of the genetic programming software. The first two types are integer types, **int** and **num**. The **num** type represents small integers in the range [0,20]. In order to integrate the instruction of the machine two types were defined for the operations: **tape** and **data**.

The reason for the definition of two separate types for the instruction is the need to combine these instructions to make correct programs. Indeed, the data operations are invalid when no data has been read from the tape. And for the same reason tape operations make no sense if the data is not processed afterwards.

[4] min (stride=1) changes the series 2_5_3_2_5_6_7 to -2_1_-1_1_2_3 + 4 (the average)

[5] pec The symbols 1_123_250 are replaced by (3,4)_(6,59)_(13,58). The last couple allows to represent 250 with two smaller numbers ($2^7 + 2^6 + 58$) where as $\log_2$ would give the couple (7,122).

4.5 Function and terminal set

Here we present the function and terminal set provided to the genetic programming system. Apart from the operations provided by the virtual machine a few other functionalities have been introduced.

The programs one can evolve consist of series of tape operations. In order to obtain those series a special function called SEQ has been used. The SEQ function will evaluate its first then its second arguments. Arguments which in turn can be other SEQ functions or real operations on the data.

The ERC_INT and ERC_NUM are used to create ephemeral random constants. These values are either used for the stride parameter expected by some data operations or for the number of symbols to read from tape using **load**. Related to this is the terminal **END**. This terminal returns the number of symbols remaining on the input tape. The introduction of this terminal made it possible to evolve programs that processed the entire input tape.

Table 1: Overview of the return types of the terminal and function set.

Name	ret type	type arg1	type arg2
LOAD	tape	int	data
FWD	tape	data	/
REW	tape	data	/
SEQ	tape	tape	tape
END	int	/	/
DPCM	data	num	/
INV	data	num	/
MIN	data	num	/
SUB	data	num	/
MTF	data	/	/
RAW	data	/	/
PEC	data	/	/
DIV	int	int	num
ERC_INT	int	/	/
ERC_NUM	num	/	/

4.6 Parameter settings

The experiments presented here were done using 2 populations of 500 individuals. The selection probabilities of the different genetic operations were:

- standard uniform subtree crossover 75%
- standard uniform mutation 20%
- reproduction 5%
- no depth or node count limit
- The selection strategy for the 3 genetic operators was tournament selection with a tournament size of 4

The island model requires additional parameters:

- exchange rate of 3 individuals every 10 generations
- the exchanged individuals (the ones being exported) were selected using tournament selection
- The worst individuals in destination subpopulation were replaced by the imported ones

5 Results

5.1 Tool-box and decoder

To validate the presented results a set of software tools have been implemented. The functions and terminals needed by the gp software are provided by an implementation of the virtual machine.

In order to reverse the transformation an encoding format has also been designed. When executing an instruction, the data produced by the virtual machine actually comprises a header and the processed input data. This header describes the instruction used to produce the data, the length of the data as well as possible parameters required by the instruction. In the present implementation this header is 9 bytes in size. It is important to point out that the program that produced the transformed data is encoded in the output data. The entropy used as an objective criterion in the fitness function, is thus the entropy of the data **and** the program that created it.

Of course a decoder is required to reconstruct the original data. Using the headers present in the data the different transformations can be reversed one by one. One can decode the preprocessed files and compare them with the original data with a program such as *diff*. The size of the statically linked decoder is 23534 bytes on a Linux x86 platform. Both the data and the decoder should be transmitted to obtain the original data. The gain in compression should exceed the size of the decoder.

5.2 Entropy reduction

Files from the Canterbury Corpus were used as input data and a separate preprocessor evolved for each of them. The reduction of the entropy obtained for some

of the files is given in table 2. The standard image compression image *lena*, for example, has been transformed so that is retains only 68.3% of its original entropy. And this without any loss of information.

Table 2: Comparison between the initial entropy and the entropy after preprocessing

File	original H	new H	% reduction
kennedy.xls	3.57	0.7	78.4
laptop.bmp	7.76	3.3	56.5
lena_std.ppm	7.75	5.2	31.7
mosaic.pnm	7.78	4.1	46.6
peppers.ppm	7.66	5.3	30.1

This reduction in entropy without loss of information is not impossible. The evolved preprocessor exploits the redundancy that is present in the data. This can not be modeled by the formula for the entropy given by equation 2. One would have to use conditional probabilities when computing the entropy to account for this kind of redundancy. This *higher order redundancy* is however very much there and is (partially) exposed by transforming the data. Although reducing the entropy of given data is not an easy task, it is of no immediate use as such. The idea behind reducing the entropy is to improve the compression rate. To illustrate this table 3 gives a comparison between the compression ratio of the data with and without preprocessing. The compression algorithm used here is bzip2. [6]

Table 3: Difference in compression ratio after preprocessing. Initially the lena could be compressed to 74% of its initial size. After preprocessing the compression ration is 68% of the size.

File	ratio	new ratio	% reduction
kennedy.coded	0.12	0.02	80.8
laptop.coded	0.53	0.43	19.1
lena_std.coded	0.74	0.68	7.9
mosaic.coded	0.72	0.52	28.3
peppers.coded	0.80	0.69	14.3

One can notice some difference between the gain in entropy and the gain in compression size. The gain in compressed size can be smaller than the reduction in entropy because Burrows Wheeler transform [1] used by bzip2 can to some extent model the redundancy beyond the single symbol probabilities. In other words, bzip2 already breaks the theoretical limit computed using the entropy formula in equation 2. That is why the gain is somewhat lower in this case.

[6] Bzip2 is free implementation of the Burrows-Wheeler transform. Bzip2 was invoked with the maximum compression parameter -9.

5.3 Filters

It was initially expected that, given the operations which can process chunks of data, the evolved programs would process the entire input data in a piecewise manner. That is, the programs would consist of a sequence of operations on contiguous parts of the data. The function set provided to the genetic programming framework certainly would allow for such programs to be evolved.

This was however not the case, in the first experiments only small fractions of the data were being processed. In their experiments Nordin and Banzhaf [10] have chosen to evolve separate programs for segments of fixed size, *chunking*, to avoid a similar problem.

The reason behind this phenomenon is that the entropy is a global measure over the entire output tape. Initial programs may indeed reduce the entropy. But the growth of these programs can adversely affect the initial changes to the entropy. Thereby making the results of the genetic operations like crossover less fit. This has been observed even when the initial population was seeded with full grown trees.

```
(SEQ (SEQ (SEQ (LOAD (DIV 67 2)(DPCM 1))
               (SEQ (LOAD END (DPCM 1))
                    (REW (DPCM 3))))
          (SEQ (REW (SUB 1))
               (REW (DPCM 3))))
     (SEQ (REW (SUB 1))
          (REW (DPCM 3))))
```

However with the introduction of the special **END** terminal in the set an interesting result showed up. This *dynamic* terminal returns the number of symbols left on the input tape. The evolved programs became filters. Most of the data goes through several transformations. In the example above, the program can be divided in 2 parts. First, 33 symbols are load into the machine and DPCM coded using the (LOAD (DIV 67 2) (DPCM 1)) instruction. The second part of the program processes far more symbols. The instruction (LOAD END (DPCM 1)) instruction loads *all the symbols* remaining on the input tape and DPCM codes these. Using the (REW (DPCM 3)) instruction the DPCM coded data is used again as input. This time the data is DPCM coded again but only every 3-th

symbol is being processed. The data will undergo 4 more transformation after this step. The remain transformation are in order: SUB, DPCM, SUB, DPCM.

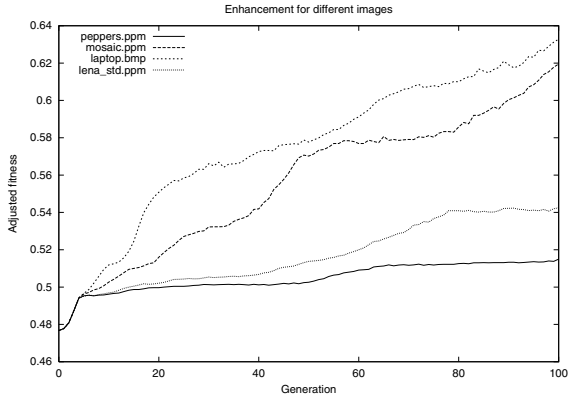


Figure 3: Evolution of the adjusted fitness of the best individual for the different data files, average of 8 runs.

6 Conclusion

This experiment presents a new approach for combining genetic programming and lossless data compression. The chosen approach develops preprocessing programs which are tailored to the data one wishes to compress. The obtained results are encouraging both in terms of gain in compression as for the computation time required to evolve the programs. One should note that, although the experiments were done using a parallel software package, the speed improvement results from the fitness function as well as the fact that we focus on preprocessors. Surprisingly the evolved programs were mostly filters although the provided functions and terminals allowed to evolve to more complex preprocessors.

References

[1] M. Burrows and D.J. Wheeler. A block-sorting lossless data compression algorithm. *Digital System Research Center research report 124*, 1994.

[2] Winzip Corporation. www.winzip.com.

[3] Canterbury Corpus. http://corpus.canterbury.ac.nz/.

[4] Karel Driesen. Compressing sparse tables using a genetic algorithm. In *Proceedings of the GRONICS'94 Student Conference, Groningen*, February 1994.

[5] I. De Falco et al. A kolmogorov complexity-base genetic programming tool for string compression. In *Proceedings Seventh International Conference on Genetic Algorithms (ICGA97)*, 1997.

[6] H. Feiel and S. Ramakrishnan. A genetic approach to color image compression. In *ACM 0-89791-850-9*, 1997.

[7] Alex Fukunaga and Andre Stechert. Evolving nonlinear models for lossless image compression with genetic programming. In *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 92–102, 1998.

[8] The Gzip homepage. www.gzip.org.

[9] Sean Luke. Code growth is not caused by introns. In *Late Breaking Papers at the 2000 Genetic and Evolutionary Conference*, 2000.

[10] Peter Nordin and Wolfgang Banzhaf. Programmatic compression of images and sound. In David B. Fogel John R. Koza, Davis E. Goldberg and Stanford University Rick L. Riolo ed., editors, *Proceedings of the First Annual Conference on Genetic Programming*, pages 354–350. CA, USA, Mit Press.

[11] J. Parent. Parallel lil-gp technical report. Vrije Universiteit Brussel, http://parallel.vub.ac.be./~johan/Projects/.

[12] B. Punch and D. Zonker. lil-gp genetic programming version 1.1 beta. michigan state university, http://garage.cps.msu.edu/software/lil-gp/index.html.

[13] P. Vitanyi. *An Introduction to Kolmogorov Complexity and Its Applications.* Addison-Wesley, Reading, Mass, 1991.

On the Search Biases of Homologous Crossover in Linear Genetic Programming and Variable-length Genetic Algorithms

Riccardo Poli
Department of Computer Science
University of Essex, UK
rpoli@essex.ac.uk

Christopher R. Stephens
Instituto de Ciencias Nucleares
UNAM, Mexico
stephens@nuclecu.unam.mx

Alden H. Wright
Computer Science Department
University of Montana, USA
wright@cs.umt.edu

Jonathan E. Rowe
School of Computer Science
University of Birmingham, UK
j.e.rowe@cs.bham.ac.uk

Abstract

With a schema-theoretic approach and experiments we study the search biases produced by GP/GA homologous crossovers when applied to linear, variable-length representations. By specialising the schema theory for homologous crossovers we show that these operators are unbiased with respect to string length. Then, we provide a fixed point for the schema evolution equations where the population presents a statistically independent distribution of primitives. This is an important step towards generalising Geiringer's theorem and the notion of linkage equilibrium.

1 INTRODUCTION

Search algorithms typically include three main steps which are iterated in succession: choosing one or multiple points of the search space the neighbourhood of which to explore further, applying *expansion operators* to obtain a new set of points, deciding what to do with the points previously visited and with the newly generated ones. For example, in the case of genetic algorithms and genetic programming, selection corresponds to the first task, crossover and mutation are the expansion operators, and the replacement strategy corresponds to the third task. Note that many more steps may be included if one looks at search algorithms at a finer level of abstraction (Poli & Logan, 1996), but this is not particularly important for the purposes of this discussion. What is important is that different algorithms will use different strategies to realise the different steps. This leads to the sampling of the search space according to different schedules. With the exception of random search, this means that some areas of the search space will be explored sooner, will be allocated more samples, or will be ignored altogether. This is what we mean by *search bias*.

Clearly the bias of an algorithm is the result of the interaction of the biases of all its components. In the case of fixed length genetic algorithms, a lot of attention has been devoted to the biases of all such components: selection, mutation and crossover, and replacement. Some of the resulting studies apply also to the case of variable-length-structure evolution. For example, the focusing effects of selection will be exactly the same, since selection is representation-independent. However, very little is known about the biases of the genetic operators used to evolve variable length representations, such as those used in linear GP (Nordin, 1994; O'Neill & Ryan, 2001) or in variable-length GAs.

Knowing the biases introduced by the operators is very important, since it leads to a deeper understanding of the search algorithm under investigation. This, in turn, allows an informed choice of operators, parameter settings and even initialisation strategies for particular problems. How can one investigate these biases? One possibility is to use carefully designed empirical studies. In the past these have shed some light on the internal dynamics of GP (e.g. on bloat (McPhee & Miller, 1995; Soule *et al.*, 1996; Langdon *et al.*, 1999)), but rarely has the evidence been general and conclusive. This is because these studies can only consider a limited number of benchmark problems, and so it is impossible to know whether, and to which extent, the observed behaviour is applicable to other problems. An alternative is to perform theoretical studies. Often these may lead to more general and precise conclusions, but they are definitely much harder and slower to carry out. Also, sometimes the complexity of the mathematics involved in these studies forces the researcher to make simplifying hypotheses which may limit the explanatory power of the results.

A class of recent theoretical results which require very few, if any, simplifications goes under the name of *exact schema theorems*.[1] These provide probabilistic models (the schema evolution equations) of the expected behaviour of a GA or a GP system over one, or, under certain assumptions, multiple generations (Poli, 2001*b*; Langdon & Poli,

[1] The word "exact" refers to the fact that, unlike earlier results, these theorems provide an exact value, rather than a lower bound, for the expected number of individuals in a schema in the next generation.

2002). The main advantage of these exact models is that they provide a natural way of coarse graining the huge number of degrees of freedom present in a genetic algorithm (Stephens & Waelbroeck, 1997). Exact schema theorems have recently become available for fixed-length GAs with one-point crossover and mutation (Stephens & Waelbroeck, 1997; Stephens & Waelbroeck, 1999), and general homologous crossover and mutation (Vose & Wright, 2001; Stephens, 2001). Even more recent is the development of exact schema theorems for variable-length GAs, linear GP and tree-based GP. These cover a variety of crossover and mutation operators including one-point crossover (Poli, 2000; Poli, 2001*b*), subtree-swapping crossovers (Poli, 2001*a*; Poli & McPhee, 2001*b*; McPhee & Poli, 2001), different types of subtree mutation and headless chicken crossover (Poli & McPhee, 2001*a*; McPhee *et al.*, 2001), and homologous crossovers (Poli & McPhee, 2001*c*).

These exact models can be used to understand an evolutionary system and study its behaviour in two different ways. This can be done either through simulation (i.e., by numerically iterating the equations) or through mathematical analysis. Although exact GP schema equations have become available only very recently, early studies indicate their usefulness, for example, in providing a deeper understanding of emergent phenomena such as bloat (Poli & McPhee, 2001*b*; McPhee & Poli, 2001). Also, in general, as indicated above the availability of exact models for different operators allows a formal study of the biases of those operators. Steps forward in this direction have recently been made in (Poli & McPhee, 2001*b*; McPhee & Poli, 2001; McPhee *et al.*, 2001), where a class of Gamma program-length distributions has been shown to represent a natural attractor for variable-length linear systems under GP subtree crossover and in (Poli *et al.*, 2002) where we have extended the study to other biases of subtree crossover.

In this paper we study the biases of the whole class of homologous GP crossover operators for the case of linear GP and variable-length GAs. These are a set of operators, including GP one-point crossover (Poli & Langdon, 1997) and GP uniform crossover (Poli & Langdon, 1998*a*), where the offspring are created preserving the position of the genetic material taken from the parents. These operators are important because they are the natural generalisation of the corresponding GA operators. So, the theory presented here is a generalisation of corresponding GA theory.

The paper is organised as follows. We start by providing some background information on the exact GP schema theory for homologous crossover and Geiringer's theorem in Sections 2 and 3. Then, we simplify the theory for the case of linear, but variable-length, structures in Section 4, and show that homologous crossover is totally unbiased with respect to string length (Section 5). In Section 6 we provide a fixed point for the schema evolution equations which is a first step towards generalising Geiringer's theorem (Geiringer, 1944) and the notion of linkage equilibrium, which, until now, were applicable only to fixed-length representations. The fixed point and some experimental evidence (reported in Section 7) indicate the presence of a bias which pushes the population towards a statistically independent distributions of primitives, as discussed in Section 8, where we also draw some conclusions.

2 SCHEMA THEORY BACKGROUND

Schemata are sets of points in a search space sharing some syntactic feature. For example, for GAs operating on binary strings the syntactic representation of a schema is usually a string of symbols from the alphabet {0,1,*}, where the character * is interpreted as a "don't care" symbol. Typically schema theorems are descriptions of how the number of members of the population belonging to a schema vary over time. If $\alpha(H,t)$ denotes the probability that at time t a newly created individual samples (or matches) the schema H, which we term the *total transmission probability* of H, then an exact schema theorem for a generational system is simply

$$E[m(H,t+1)] = M\,\alpha(H,t),$$

where M is the population size, $m(H,t+1)$ is the number of individuals sampling H at generation $t+1$ and $E[\cdot]$ is the expectation operator. Holland's (Holland, 1975) and other (e.g. (Poli & Langdon, 1998*b*)) worst-case-scenario schema theories normally provide a lower bound for $\alpha(H,t)$ or, equivalently, for $E[m(H,t+1)]$. However, recently exact schema theorems (Stephens & Waelbroeck, 1997; Stephens & Waelbroeck, 1999; Poli, 2000; Poli, 2001*b*; Poli, 2001*a*; Poli & McPhee, 2001*b*; McPhee & Poli, 2001; Poli & McPhee, 2001*a*; McPhee *et al.*, 2001; Stephens, 2001; Poli & McPhee, 2001*c*) which provide the exact value for $\alpha(H,t)$ have become available for GAs and GP with a variety of operators. In the remainder of this section we will introduce the various elements which are necessary to understand the exact schema equations for GP homologous crossover, since these will be the starting point for the new results in this paper.

Let us start from our definition of schema for GP. Syntactically a GP schema is a tree composed of functions from the set $\mathcal{F}\cup\{=\}$ and terminals from the set $\mathcal{T}\cup\{=\}$, where $\mathcal{F}$ and $\mathcal{T}$ are the function and terminal sets used in a GP run. The primitive $=$ is a "don't care" symbol which stands for a *single* terminal or function. A schema H represents the set of all programs having the same shape as H and the same non-$=$ nodes as H. Particularly important for the GP schema theory are schemata containing "don't care" symbols only, since they represent all the programs of a particular shape. Let G_1, G_2, $\cdots$ be an enumeration of such shape-representing schemata.

In GP homologous crossovers the offspring are created by exchanging genetic material (nodes and subtrees) taken

from the same position in the parents trees. To account for the possible structural diversity of the two parents, the selection of the nodes and the roots of the subtrees to swap are constrained to belong to the *common region*. This is the largest rooted region where the two parent trees have the same topology.

In order to define more precisely how GP homologous crossovers work, we start by providing a formal definition of common region. The common region between two generic trees h_1 and h_2 is the set $C(h_1, h_2) = \{(d, i)|\mathcal{C}(d, i, h_1, h_2)\}$, where (d, i) is a pair of coordinates in a Cartesian node reference system (Poli, 2001*a*; Poli & McPhee, 2001*c*). The predicate $\mathcal{C}(d, i, h_1, h_2)$ is true if $(d, i) = (0, 0)$ (i.e., if (d, i) is the root node). It also true if $A(d-1, i', h_1) = A(d-1, i', h_2) \neq 0$ and $\mathcal{C}(d-1, i', h_1, h_2)$ is true, where $A(d, i, h)$ returns the arity of the node at coordinates (d, i) in h, $i' = \lfloor i/a_{\max} \rfloor$, $a_{\max}$ is the maximum arity of the functions in the function set, and $\lfloor \cdot \rfloor$ is the integer-part function. The predicate is false otherwise. The notion of common region can be applied to schemata, too.

To complete our formal description of the class of GP homologous crossovers, we need to extend to GP the notions of crossover masks and recombination distributions used in genetics (Geiringer, 1944) and in the GA literature (Booker, 1992; Altenberg, 1995; Spears, 2000). Let us first briefly recall the definition of these notions for a GA operating on fixed-length binary strings. In this case a crossover mask is simply a binary string. When crossover is executed, the bits of the offspring corresponding to the 1's in the mask will be taken from one parent, those corresponding to 0's from the other parent. If the GA operates on strings of length N, then 2^N different crossover masks are possible. If, for each mask i, one defines a probability, p_i, that the mask is selected for crossover, then it is easy to see how different crossover operators can simply be interpreted as different ways of choosing the probability distribution p_i. The distribution p_i is called a *recombination distribution*.

For the more general case of GP and variable-length GAs, for any given common region c we can define a set of *GP crossover masks*, χ_c, which contains all different trees with the same size and shape as c which can be built with nodes labelled 0 and 1 (Poli & McPhee, 2001*c*; Poli *et al.*, 2001). Each crossover mask represents one of the ways in which one could generate an offspring through crossover: nodes of the offspring corresponding to internal 1's in the mask will be taken from the first parent, nodes corresponding to internal 0's from the second parent, subtrees of the first parent whose root corresponds to leaves labelled with a 1 in the mask will be transferred to the same position in the offspring, and, finally, subtrees of the second parent whose root corresponds to leaves labelled with a 0 in the mask will be transferred to the same position in the offspring. The *GP recombination distribution* p_l^c gives the probability that, for a given common region c, crossover mask l will be chosen from the set χ_c. Each GP homologous crossover is characterised by a different recombination distribution. Since the size and shape of the common region can be inferred from the mask l, in the following we will often omit the superscript c from p_l^c.

Finally, before we introduce the exact schema equation for GP homologous crossover developed in (Poli & McPhee, 2001*c*) we need to define the notion of hyperschema. A *GP hyperschema* is a rooted tree composed of internal nodes from $\mathcal{F} \cup \{=\}$ and leaves from $\mathcal{T} \cup \{=, \#\}$. Again, = is a "don't care" symbols which stands for exactly one node, while # stands for any valid subtree. In the theory we use hyperschemata to represent the characteristics the parents must have to produce instances of a particular schema of interest.

The exact schema equations for GP with homologous crossover are

$$\alpha(H, t) = (1 - p_{xo})p(H, t) + p_{xo}\alpha_{xo}(H, t) \qquad (1)$$

where

$$\alpha_{xo}(H, t) = \sum_j \sum_k \sum_{l \in \chi_{C(G_j, G_k)}} p_l \; p(\Gamma(H, l) \cap G_j, t) p(\Gamma(H, \bar{l}) \cap G_k, t), \qquad (2)$$

p_{xo} is the crossover probability, $p(H, t)$ is the selection probability of the schema H and $\bar{l}$ is the complement of the GP crossover mask l (i.e. it is a tree with the same structure as l but with the 0's and 1's swapped). $\Gamma(H, l)$ is defined to be the empty set if l contains any node not in H. Otherwise it is the hyperschema obtained by replacing certain nodes in H with either = or # nodes: (1) if a node in H corresponds to (i.e., has the same coordinates as) a non-leaf node in l that is labelled with a 0, then that node in H is replaced with a =, (2) if a node in H corresponds to a leaf node in l that is labelled with a 0, then it is replaced with a #, (3) all other nodes in H are left unchanged.

As discussed in (Poli & McPhee, 2001*c*), it is possible to show that, in the absence of mutation, Equations 1 and 2 generalise and refine not only approximate GA and GP schema theorems (Holland, 1975; Poli & Langdon, 1997; Poli & Langdon, 1998*b*) but also more recent exact schema theorems (Stephens & Waelbroeck, 1997; Stephens & Waelbroeck, 1999; Poli, 2000; Stephens, 2001).

In the following we will use a slightly different form for Equation 2 which exploits the symmetries in the process of selection of the parent programs. This can be obtained by dividing each set of crossover masks $\chi_{C(G_j, G_k)}$ into two non-overlapping sets $\chi'_{C(G_j, G_k)}$ and $\bar{\chi}'_{C(G_j, G_k)}$ such that for each mask $x \in \chi'_{C(G_j, G_k)}$, there is a mask $y \in \bar{\chi}'_{C(G_j, G_k)}$ such that $y = \bar{x}$, and *vice versa*. Then, by reordering the terms, it is easy to prove that:

Theorem 1.

$$\alpha_{xo}(H,t) = \sum_j \sum_k \sum_{l \in \chi'_{C(G_j,G_k)}} (p_l + p_{\bar{l}}) \; p(\Gamma(H,l) \cap G_j, t) p(\Gamma(H,\bar{l}) \cap G_k, t) \quad (3)$$

3 GEIRINGER'S THEOREM

In this section we briefly introduce Geiringer's theorem (Geiringer, 1944), an important result with implications both for natural population genetics and evolutionary algorithms (Booker, 1992; Spears, 2000). Geiringer's theorem indicates that, in a population of fixed-length chromosomes repeatedly undergoing crossover (in the absence of mutation and selective pressure), the probability of finding a generic string $h_1 h_2 \cdots h_N$ approaches a limit distribution which is only dependent on the distribution of the alleles h_1, h_2, etc. in the initial generation. More precisely, if $\Phi(h_1 h_2 \cdots h_N, t)$ is the proportion of individuals of type $h_1 h_2 \cdots h_N$ at generation t and $\Phi(h_i, t)$ is the proportion of individuals carrying allele h_i then

$$\lim_{t \to \infty} \Phi(h_1 h_2 \cdots h_N, t) = \prod_{i=1}^{N} \Phi(h_i, 0). \quad (4)$$

This result is valid for all homologous crossover operators which allow any two loci to be separated by recombination. Strictly speaking the result is valid only for infinite populations.

If one interprets $\Phi(h_1 h_2 \cdots h_N, t)$ as a probability distribution of the possible strings in the population, we can interpret Equation 4 as saying that such a distribution is converging towards independence. When, at a particular generation t, the frequency of any string in a population $\Phi(h_1 h_2 \cdots h_N, t)$ equals $\prod_{i=1}^{N} \Phi(h_i, t)$, the population is said to be in *linkage equilibrium* or *Robbins' proportions*.

It is trivial to generalise Geiringer's theorem to obtain the expected fixed-point proportion of a generic linear fixed-length GA schema H for a population undergoing crossover only:

$$\lim_{t \to \infty} \Phi(H, t) = \prod_{i \in \Delta(H)} \Phi(*^{i-1} h_i *^{N-i}, 0), \quad (5)$$

where $\Delta(H)$ is the set of indices of the defining symbols in H, h_i is one such defining symbols and we used the power notation x^y to mean x repeated y times. (Note that $\Phi(*^{i-1} h_i *^{N-i}, t)$ coincides with $\Phi(h_i, t)$.)

4 EXACT SCHEMA THEORY FOR LINEAR STRUCTURES

As indicated in Section 1 in this paper we will consider the biases of the homologous crossovers in the case of variable-size linear representations. We start by specialising Equation 3 to this case.

When only unary functions are used in tree-based GP, schemata (and programs) can only take the form $(h_1(h_2(h_3....(h_{N-1}h_N)....)))$ where $N > 0$, $h_i \in \mathcal{F} \cup \{=\}$ for $1 \leq i < N$, and $h_N \in \mathcal{T} \cup \{=\}$. Therefore, they can be written unambiguously as strings of symbols of the form $h_1 h_2 h_3 .... h_{N-1} h_N$. It should be noted that these strings of symbols do not have to be necessarily interpreted as representing programs. If one uses a special terminal set $\mathcal{T}$ including only one terminal, say `EOR` (for End Of Representation), which will be ignored when the representation is interpreted, then strings of the form $h_1 h_2 h_3 .... h_{N-1} h_N$ can be interpreted as chromosomes of length $N-1$ (since h_N can only be `EOR`). So, if $\mathcal{F} = \{0, 1\}$, where 0 and 1 are "unary functions",[2] our GP system will explore the space of variable length binary strings. If instead $\mathcal{F}$ includes the "unary functions" $\{$`ADD R0 R1`, `MUL R0 R1`, $\ldots\}$, then our tree-based GP system explores the same search space as a machine-code GP system with the same primitive set (Nordin & Banzhaf, 1995). So, in general our specialisation of Equation 3 will be valid for variable-length GAs and linear GP.

In the specialisation to the linear case we replace the "don't care" symbol "=" with the more standard symbol "*". Also, as we did previously, we represent repeated symbols in a string using the power notation. Since in this case all trees are linear, the space of program shapes can be enumerated by $\{G_n\}$ where G_n is $*^n$ for $n > 0$. Given this, the common region between shapes G_j and G_k is simply the shorter of the two schemata, i.e. $C(G_j, G_k) = G_{j \downarrow k} = *^{j \downarrow k}$ where the operator $\downarrow$ returns the minimum of its two arguments. Therefore, the set of crossover masks in the common region, $\chi_{C(G_j,G_k)} = \chi_{*^{j \downarrow k}}$, can be identified with the set $\{0,1\}^{j \downarrow k}$. Below we will use $N(l)$ to denote the length of mask l, and the notation l_i to indicate the i-th element of bitmask l. We will also use the operators

$$a \bullet b = \begin{cases} a & \text{if } b = 1 \\ * & \text{otherwise,} \end{cases}$$

$$a_1 a_2 \cdots \circ b = \begin{cases} a_1 a_2 \cdots & \text{if } b = 1 \\ \# & \text{otherwise,} \end{cases}$$

where a and b are bits, $a_1 a_2 \cdots$ is a bit string and # stands for any sequence of at least one primitive. With this notation, it is easy to show that, in a linear representation, if $N(l) > N$ then $\Gamma(H, l)$ is the empty set and $\Gamma(H, l) \cap G_j = \emptyset \cap *^j = \emptyset$. If $N(l) \leq N$, $\Gamma(H, l)$ is

$$(h_1 \bullet l_1) \cdots (h_{N(l)-1} \bullet l_{N(l)-1})(h_{N(l)} \cdots h_N \circ l_{N(l)})$$

and

[2] We are not interested in the output of these functions, but simply in their topological organisation within the individual.

$$\Gamma(H,l)\cap G_j = \Gamma(H,l)\cap *^j = \quad (6)$$
$$\begin{cases} (h_1 \bullet l_1)\cdots(h_{N(l)-1}\bullet l_{N(l)-1})*^{j-N(l)+1} \\ \qquad \text{if } j \geq N(l) \text{ and } l_{N(l)}=0, \\ (h_1 \bullet l_1)\cdots(h_{N(l)-1}\bullet l_{N(l)-1})h_{N(l)}\cdots h_N \\ \qquad \text{if } j = N \text{ and } l_{N(l)}=1, \\ \emptyset \qquad \text{otherwise.} \end{cases}$$

Thus, $p(\Gamma(H,l)\cap G_j,t)=0$ for all $j\neq N$ for all the masks l for which $l_{N(l)}=1$. So, if we choose $\chi'_{C(G_j,G_k)} = \{0,1\}^{j\downarrow k-1}\times\{1\}$, in Equation 3 only the terms for $j=N$ can be non-zero. Using this simplification and the previous results, one can transform Equation 3 into:

Theorem 2. *If* $\chi'_{*^{N\downarrow k}} = \{0,1\}^{j\downarrow k-1}\times\{1\}$*, then*

$$\alpha_{xo}(h_1...h_N,t) = \sum_{k>0}\sum_{l\in\chi'_{*^{N\downarrow k}}}(p_l+p_{\bar{l}}) \quad (7)$$
$$p((h_1\bullet l_1)\cdots(h_{N\downarrow k-1}\bullet l_{N\downarrow k-1})h_{N\downarrow k}\cdots h_N,t)$$
$$p((h_1\bullet \bar{l}_1)\cdots(h_{N\downarrow k-1}\bullet \bar{l}_{N\downarrow k-1})*^{k-N\downarrow k+1},t).$$

5 LENGTH EVOLUTION

Equation 7 can be used to study, among other things, the evolution of size in linear GP/GA systems. This is because it can be specialised to schemata of the form $*^N$ obtaining:

$$\begin{aligned}\alpha_{xo}(*^N,t) &= \sum_{k>0}\sum_{l\in\chi'_{*^{N\downarrow k}}}(p_l+p_{\bar{l}})\,p(*^N,t)p(*^k,t) \\ &= p(*^N,t)\sum_{k>0}p(*^k,t)\sum_{l\in\chi'_{*^{N\downarrow k}}}(p_l+p_{\bar{l}})\,.\end{aligned}$$

But $\sum_{l\in\chi'_{*^{N\downarrow k}}}(p_l+p_{\bar{l}})=1$ and $\sum_{k>0}p(*^k,t)=1$, so:

Theorem 3.
$$\alpha_{xo}(*^N,t)=p(*^N,t). \quad (8)$$

This result indicates that in linear representations length evolves under homologous crossovers as if selection only was acting. So, homologous crossovers are totally unbiased with respect to program length. The lack of length bias of homologous crossovers is made particularly clear if one assumes a flat fitness landscape in which $p(H,t)=\Phi(H,t)$ for all H. In these conditions all the dynamics in the system must be caused by crossover or by sampling effects. In the infinite population limit, the total transmission probability $\alpha(H,t)$ can also be interpreted as the proportion of individuals in the population in H at generation $t+1$, $\Phi(H,t+1)$. So, for an infinite population and a flat landscape Equation 8 becomes $\Phi(*^N,t+1)=\Phi(*^N,t)$, whereby

Corollary 4. *For a flat landscape, an infinite population and any* $t>0$

$$\Phi(*^N,t)=\Phi(*^N,0).$$

This equation is important because it shows that when a homologous crossover alone is acting, any initial distribution of lengths is a fixed point length distribution for the system.

6 EXTENSION OF GEIRINGER'S THEOREM

A full extension of Geiringer's theorem to linear, variable-length structures and homologous GP crossover would require two steps: (a) proving that, in the absence of mutation and of selective pressure and for an infinite population, a distribution $\Phi(h_1h_2\cdots h_N,t)$, where the alleles/primitives can be considered independent stochastic variables, is a fixed point, and (b) showing that the system indeed moves towards that fixed point. In this paper we prove (a) mathematically and provide experimental evidence for (b).

Theorem 5. *A fixed point distribution for the proportion of a linear, variable-length schema* $h_1h_2\cdots h_N$ *under homologous crossover for an infinite population on a flat fitness landscape in the absence of mutation is*

$$\Phi(h_1h_2\cdots h_N,t)=\Phi(*^{N-1}h_N,0)\prod_{i=1}^{N-1}\frac{\Phi(*^{i-1}h_i\#,0)}{\Phi(*^i\#,0)}, \quad (9)$$

where

$$\Phi(*^{i-1}h_i\#,0)=\sum_{n>0}\Phi(*^{i-1}h_i*^n,0)$$

and

$$\Phi(*^i\#,0)=\sum_{n>0}\Phi(*^{i+n},0).$$

Proof. Since the fitness landscape is flat, $p(H,t)=\Phi(H,t)$ for any schema. Also, because the population is infinite, $\alpha(H,t)=\Phi(H,t+1)$. So, combining Equations 1 and 7 yields

$$\begin{aligned}&\Phi(h_1...h_N,t+1) \quad (10)\\ &= (1-p_{xo})\Phi(h_1...h_N,t)+p_{xo}\sum_{k>0}\sum_{l\in\chi'_{*^{N\downarrow k}}}(p_l+p_{\bar{l}}) \\ &\times\ \Phi((h_1\bullet l_1)\cdots(h_{N\downarrow k-1}\bullet l_{N\downarrow k-1})h_{N\downarrow k}\cdots h_N,t) \\ &\times\ \Phi((h_1\bullet \bar{l}_1)\cdots(h_{N\downarrow k-1}\bullet \bar{l}_{N\downarrow k-1})*^{k-N\downarrow k+1},t).\end{aligned}$$

We prove that Equation 9 represents a fixed point for Equation 10 by substituting the former in the latter and reordering the terms, obtaining

$$\begin{aligned}&\Phi(h_1...h_N,t+1) \\ &= (1-p_{xo})\Phi(*^{N-1}h_N,0)\prod_{i=1}^{N-1}\frac{\Phi(*^{i-1}h_i\#,0)}{\Phi(*^i\#,0)} \\ &+\ p_{xo}\Phi(*^{N-1}h_N,0)\sum_{k>0}\Phi(*^k,0)\end{aligned}$$

$$\times \sum_{l \in \chi'_{*^{N\downarrow k}}} (p_l + p_{\bar{l}})$$
$$\times \prod_{i=1}^{N\downarrow k-1} \left(\frac{\Phi(*^{i-1}(h_i \bullet l_i)\#, 0)\Phi(*^{i-1}(h_i \bullet \bar{l}_i)\#, 0)}{(\Phi(*^i\#, 0))^2} \right)$$
$$\times \prod_{i=N\downarrow k}^{N-1} \frac{\Phi(*^{i-1}h_i\#, 0)}{\Phi(*^i\#, 0)}.$$

Whatever the value of bit l_i in the mask, either $h_i \bullet l_i = h_i$ and $h_i \bullet \bar{l}_i = *$, or $h_i \bullet l_i = *$ and $h_i \bullet \bar{l}_i = h_i$. In either case $\Phi(*^{i-1}(h_i \bullet l_i)\#, 0)\Phi(*^{i-1}(h_i \bullet \bar{l}_i)\#, 0) = \Phi(*^{i-1}h_i\#, 0)\Phi(*^i\#, 0)$. So, after reordering the terms, we obtain:

$$\begin{aligned}
&\Phi(h_1...h_N, t+1) \\
&= (1-p_{xo})\Phi(*^{N-1}h_N, 0) \prod_{i=1}^{N-1} \frac{\Phi(*^{i-1}h_i\#, 0)}{\Phi(*^i\#, 0)} \\
&+ p_{xo}\Phi(*^{N-1}h_N, 0) \prod_{i=1}^{N-1} \frac{\Phi(*^{i-1}h_i\#, 0)}{\Phi(*^i\#, 0)} \\
&\times \sum_{k>0} \Phi(*^k, 0) \sum_{l \in \chi'_{*^{N\downarrow k}}} (p_l + p_{\bar{l}}) \\
&= \Phi(*^{N-1}h_N, 0) \prod_{i=1}^{N-1} \frac{\Phi(*^{i-1}h_i\#, 0)}{\Phi(*^i\#, 0)} \qquad \square
\end{aligned}$$

It is important to note that, although Equation 9 provides a family of fixed points, this does not prove rigorously that any population will always converge to one of them. Proving this is complex and requires much more space than is available for this conference. We will provide the proof in a future more extended publication. Instead, in the following section we will describe experimental results which strongly suggest that indeed populations move toward an independent allele/primitive distribution.

7 EXPERIMENTAL RESULTS

In order to check the theoretical results in this paper we set up a population of variable length strings consisting of 1,000,000 individuals. All individuals had the same terminal allele, 0, while two types of non-terminal alleles were used: alleles of type 0 and alleles of type 1. The majority of alleles were of type 0 and represented a "background" against which alleles of type 1 (the "contrast medium" we used to study the dynamics of non-terminal alleles) could be more easily traced. Initially, alleles of type 1 occupied all the non-terminal loci of strings of a given length only (which was varied between experiments). The terminal locus of those strings was occupied by an allele of type 0. All other terminal and non-terminal alleles in the population were of type 0.

In our experiments we used two different initial length distributions: a Gamma distribution with mean 10.5, and a uniform distribution with lengths between 1 and 20. Each population was run for 100 generations. The system was a generational GP/GA system with either one-point crossover or uniform crossover (applied with 100% probability) and a flat fitness landscape. One-point crossover is a homologous crossover operator which, for variable length strings, is characterised by the recombination distribution

$$p_l^{*^n} = \begin{cases} 1/n & \text{if } l \in \{0^n, 10^{n-1}, 110^{n-2}, \ldots, 1^{n-1}0\}, \\ 0 & \text{otherwise.} \end{cases}$$

Uniform crossover has the recombination distribution $p_l^{*^n} = 2^{-n}$.

Multiple independent runs were not required since the population size was sufficiently large to remove any significant statistical variability and therefore to approximate the infinite-population behaviour (for each program length we had tens of thousands of individuals on average).

We start by checking what happens to the length distribution over time. Figure 1 shows that the distribution of program length is at a fixed point when the population is initialised using either a uniform length distribution or a discrete Gamma distribution. This corroborates our finding that any length distribution is a fixed point (Corollary 4). Note that the small variations in the plots are due to genetic drift (i.e. a finite population effect).

Let us now consider the allele dynamics. Figure 2 shows how the distribution of alleles of type 1 varies over a number of generations in a population initialised with the uniform length distribution. In Figure 2(a) only strings of length 2 included alleles of type 1 (in locus 1). However, under one-point crossover within a few generations (see Figures 2(b) and (c)) the relative proportion of strings with a 1 in locus 1 reached the equilibrium value predicted by applying Equation 9 to the schema $1*^{N-1}$:

$$\begin{aligned}
\lim_{t\to\infty} \frac{\Phi(1*^{N-1}, t)}{\Phi(*^N, 0)} &= \frac{\Phi(1\#, 0)}{\Phi(*\#, 0)} \prod_{i=2}^{N-1} \frac{\Phi(*^{i-1}*\#, 0)}{\Phi(*^i\#, 0)} \\
&\approx \frac{\frac{1}{20}}{\frac{19}{20}} \times \prod_{i=2}^{N-1} 1 = \frac{1}{19} \approx 0.0526.
\end{aligned}$$

The same asymptotic value is approached when only strings of length 10 included alleles of type 1 at generation 0, as shown in Figures 2(d)–(g). Qualitatively the behaviour of other loci is the same, but the asymptotic values reached are slightly different, which is predicted by Theorem 5.

Figure 2 reveals that the speed with which alleles in different loci approach their asymptotic value varies. While alleles in locus 1 move quickly towards their fixed point, the convergence speed decreases as the locus position in-

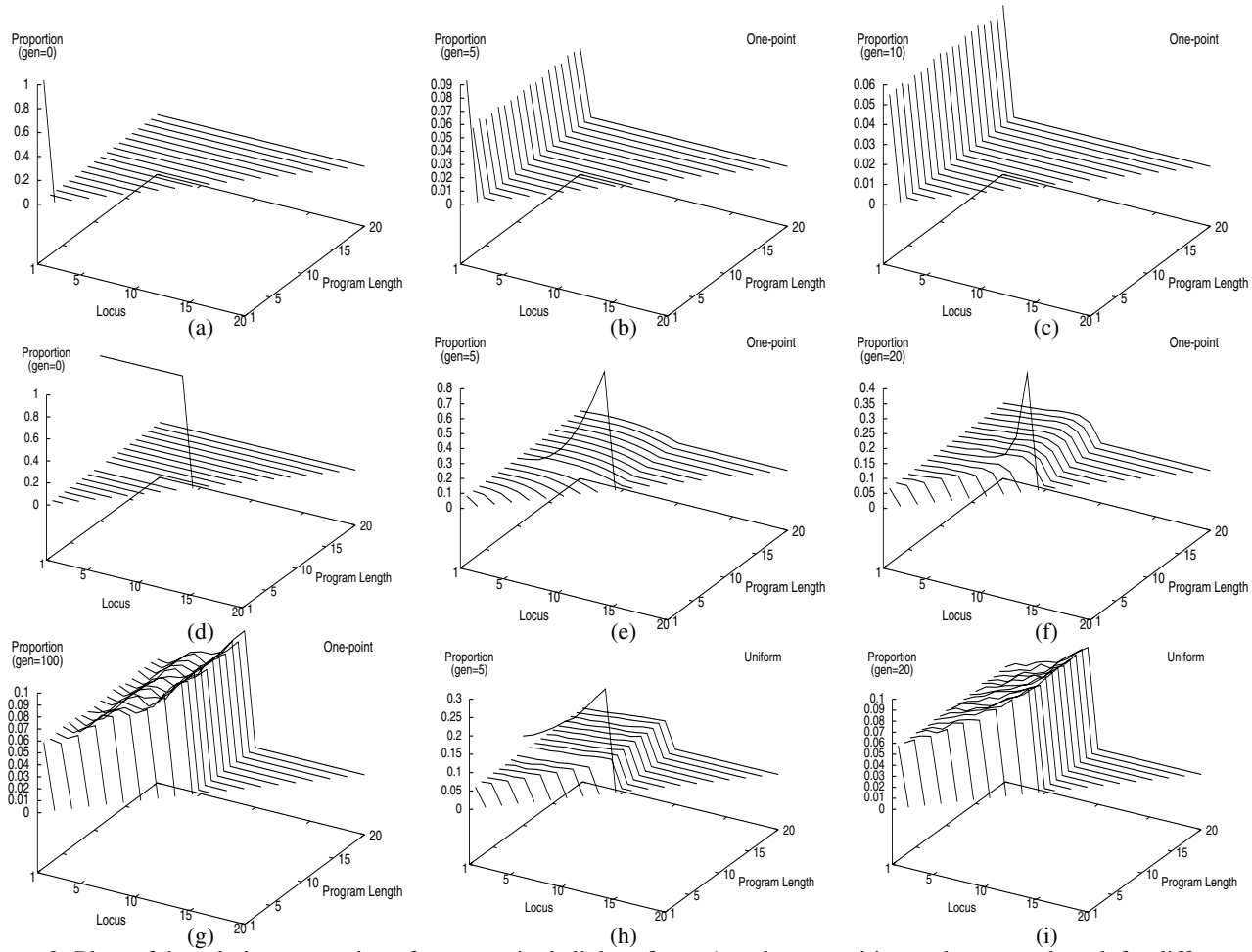


Figure 2: Plots of the relative proportion of non-terminal alleles of type 1 vs. locus position and program length for different generations under different crossover operators. The population was initialised with a uniform length distribution, where only programs of length ℓ contained non-terminal alleles of type 1. The value of ℓ was 2 in (a)–(c) and 10 in (d)–(i). One-point crossover was used in (b), (c), (e), (f) and (g), uniform crossover in (h) and (i).

creases. This is due to the fact that alleles occupying non-terminal loci that are present in a large number of individuals will be swapped more frequently than alleles occupying loci present in a small number of individuals.

The behaviour of uniform crossover is almost identical. Analysis of our results revealed that the only difference is the speed with which alleles in different loci approach their asymptotic value. Uniform crossover mixes alleles more quickly as can easily be seen, for example, by comparing Figures 2(e) and (f) with Figures 2(h) and (i), respectively.

To further verify that under homologous crossover the population tends towards an independent allele distribution, we performed an experiment with exactly the same set up as in Figures 2(h) and (i) but this time we kept track of the co-occurrence of pairs of non-terminal alleles within the class of programs of length 10. So, for each generation we obtained a set of four 9×9 co-occurrence frequency matrices, one for each possible choice of a pair of the non-terminal alleles 0 and 1. An element at position (r, c) of the co-occurrence matrix for non-terminal alleles a and b, represented the average number of times allele a was present in locus r while at the same time allele b was present in locus c in strings of length 10. Once normalised by the total number of strings of length 10, the diagonal elements of the 0/0 and 1/1 matrices represent the proportions of alleles of type 0 and 1, respectively, present at each locus.

For all allele pairs the co-occurrence matrices tended to those predicted by the theory. For example, for the allele pair 0/0 the theoretical values for the off-diagonal elements can be calculated using the following equation (obtained from Equation 9)

$$\lim_{t\to\infty} \frac{\Phi(*^a 0\, *^b\, 0 *^{10-a-b-2}, t)}{\Phi(*^{10}, 0)}$$

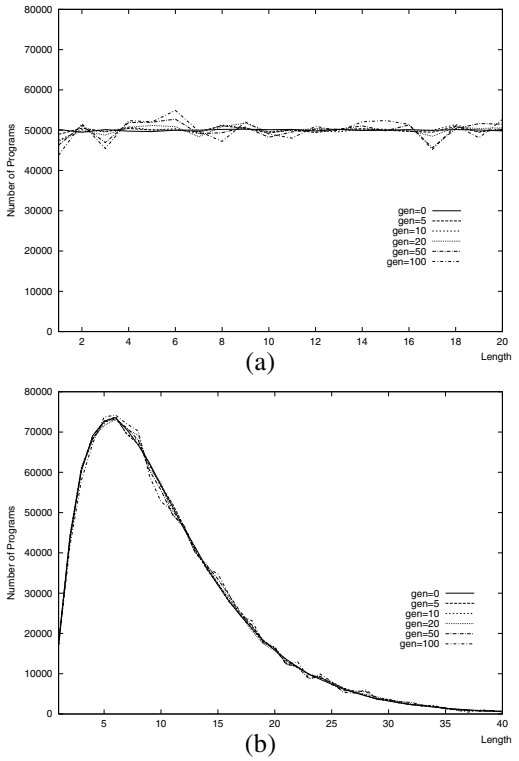


Figure 1: Plots of the number of programs vs. program length for different generations for populations initialised with a uniform (a) and a Gamma (b) length distribution.

$$= \frac{\Phi(*^a 0\#, 0)}{\Phi(*^{a+1}\#, 0)} \times \frac{\Phi(*^{a+b+1}0\#, 0)}{\Phi(*^{a+b+2}\#, 0)}$$
$$\approx \frac{\frac{18-a}{20}}{\frac{19-a}{20}} \times \frac{\frac{17-a-b}{20}}{\frac{18-a-b}{20}} = \frac{18-a}{19-a} \times \frac{17-a-b}{18-a-b}.$$

These values are extremely close to the frequency of 0/0 co-occurrence measured at generation 100 in our runs, the root mean square error between the above-diagonal elements being 0.004 (note that the co-occurrence matrix is symmetric). This is a tiny error considering that all frequencies were bigger than 0.8.

8 DISCUSSION AND CONCLUSIONS

Characterisations of the genetic biases of the operators (such as the ones offered in this paper and in (Poli *et al.*, 2002)) are important because they allow the users of GP/GA systems to evaluate whether their operators provide the desired search behaviour for the system. If this is not the case, then the knowledge of the search biases of other operators allows for an informed choice for an alternative.

Here we have focused our attention on the biases of homologous crossovers with respect to length and allele distribution in a population of variable length linear structures and presented theoretical results describing the asymptotic behaviour for a GP/GA system evolving in a flat fitness landscape. In addition, we have provided experimental evidence that firmly corroborates the theory, showing a perfect match (within experimental errors) between the predictions of the theory based on generation 0 data and the observed length and allele frequencies at later generations.

The behaviour we have observed and characterised is simple: a) homologous crossovers are totally unbiased with respect to program length, and b) crossover shuffles the alleles present in different individuals and pushes the string distribution towards locus-wise independence.

A mixing behaviour is present in most crossover operators described in the literature on fixed length GAs. It is well known that this destroys "linkage", i.e. correlations, between different allele positions in the population. In the fixed length case the asymptotic convergence towards independence described by Geiringer's theorem is the result of the decay of correlations due to the mixing effect of crossover. Because the representation and operators considered in this paper are generalisations of the corresponding fixed-length ones, it is not so surprising to see that linear GP is also moving towards an independent fixed-point string distribution. In other words, allele mixing is the reason why the right hand side of Equation 9 is a product, like the right hand side of Equation 4. We have no reason to believe that the situation would be significantly different in tree-based GP. Because our extension of Geiringer's theorem is the result of specialising and studying the GP schema theorem's equations, it is not unlikely that in the future we will be able to provide a Geiringer-like theorem for tree-based GP.

Our theoretical results were obtained for the extreme case of infinite populations and flat fitness landscapes. So, why should these be of any relevance to finite GP/GA populations and realistic landscapes? Firstly, because the biases of homologous crossovers in the absence of selection indicate the precise way in which this type of operators would *naturally* tend to explore the search space. When selection is added, the search bias will be modified by the focusing bias of selection, but, except in cases of very strong selection, many of the features of the search bias shown on a flat landscape will be retained. Secondly, because as shown in our experiments, the results obtained with real (but large) populations match very closely the infinite population theory. For smaller populations, the theory can still be used to give short term indications of the behaviour of the system.

Acknowledgements

CRS would like to thank the University of Birmingham for a visiting Professorship and DGAPA-PAPIIT grant IN100201. RP and CRS would like to thank the Royal Society and the University of Essex for their support. Alden Wright did this work while visiting the University of Birmingham, supported by EPSRC grant GR/R47394.

References

Altenberg, L. (1995) The Schema Theorem and Price's Theorem. In *Foundations of Genetic Algorithms 3*, (Whitley, L. D. & Vose, M. D., eds), pp. 23–49 Morgan Kaufmann, Estes Park, Colorado, USA.

Booker, L. B. (1992) Recombination distributions for genetic algorithms. In *FOGA-92, Foundations of Genetic Algorithms*, Vail, Colorado.

Geiringer, H. (1944) On the probability theory of linkage in Mendelian heredity. *Annals of Mathematical Statistics,* **15** (1), 25–57.

Holland, J. (1975) *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, USA.

Langdon, W. B. & Poli, R. (2002) *Foundations of Genetic Programming*. Springer.

Langdon, W. B., Soule, T., Poli, R. & Foster, J. A. (1999) The evolution of size and shape. In *Advances in Genetic Programming 3*, (Spector, L., Langdon, W. B., O'Reilly, U.-M. & Angeline, P. J., eds),. MIT Press Cambridge, MA, USA pp. 163–190.

McPhee, N. F. & Miller, J. D. (1995) Accurate replication in genetic programming. In *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, (Eshelman, L., ed.), pp. 303–309 Morgan Kaufmann, Pittsburgh, PA, USA.

McPhee, N. F. & Poli, R. (2001) A schema theory analysis of the evolution of size in genetic programming with linear representations. In *Genetic Programming, Proceedings of EuroGP 2001* LNCS Springer-Verlag, Milan.

McPhee, N. F., Poli, R. & Rowe, J. E. (2001) A schema theory analysis of mutation size biases in genetic programming with linear representations. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001* pp. 1078–1085 IEEE Press, COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea.

Nordin, P. (1994) A compiling genetic programming system that directly manipulates the machine code. In *Advances in Genetic Programming*, (Kinnear, Jr., K. E., ed.),. MIT Press pp. 311–331.

Nordin, P. & Banzhaf, W. (1995) Complexity compression and evolution. In *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, (Eshelman, L., ed.), pp. 310–317 Morgan Kaufmann, Pittsburgh, PA, USA.

O'Neill, M. & Ryan, C. (2001) Grammatical evolution. *IEEE Transaction on Evolutionary Compuation,* **5** (4).

Poli, R. (2000) Exact schema theorem and effective fitness for GP with one-point crossover. In *Proceedings of the Genetic and Evolutionary Computation Conference*, (Whitley, D., Goldberg, D., Cantu-Paz, E., Spector, L., Parmee, I. & Beyer, H.-G., eds), pp. 469–476 Morgan Kaufmann, Las Vegas.

Poli, R. (2001*a*) General schema theory for genetic programming with subtree-swapping crossover. In *Genetic Programming, Proceedings of EuroGP 2001* LNCS Springer-Verlag, Milan.

Poli, R. (2001*b*) Exact schema theory for genetic programming and variable-length genetic algorithms with one-point crossover. *Genetic Programming and Evolvable Machines,* **2** (2), 123–163.

Poli, R. & Langdon, W. B. (1997) A new schema theory for genetic programming with one-point crossover and point mutation. In *Genetic Programming 1997: Proceedings of the Second Annual Conference*, (Koza, J. R., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M., Iba, H. & Riolo, R. L., eds), pp. 278–285 Morgan Kaufmann, Stanford University, CA, USA.

Poli, R. & Langdon, W. B. (1998*a*) On the search properties of different crossover operators in genetic programming. In *Genetic Programming 1998: Proceedings of the Third Annual Conference*, (Koza, J. R., Banzhaf, W., Chellapilla, K., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M. H., Goldberg, D. E., Iba, H. & Riolo, R., eds), pp. 293–301 Morgan Kaufmann, University of Wisconsin, Madison, Wisconsin, USA.

Poli, R. & Langdon, W. B. (1998*b*) Schema theory for genetic programming with one-point crossover and point mutation. *Evolutionary Computation,* **6** (3), 231–252.

Poli, R. & Logan, B. (1996) The evolutionary computation cookbook: recipes for designing new algorithms. In *Proceedings of the Second Online Workshop on Evolutionary Computation*, Nagoya, Japan.

Poli, R. & McPhee, N. F. (2001*a*) Exact GP schema theory for headless chicken crossover and subtree mutation. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC 2001*, Seoul, Korea.

Poli, R. & McPhee, N. F. (2001*b*) Exact schema theorems for GP with one-point and standard crossover operating on linear structures and their application to the study of the evolution of size. In *Genetic Programming, Proceedings of EuroGP 2001* LNCS Springer-Verlag, Milan.

Poli, R. & McPhee, N. F. (2001*c*) Exact schema theory for GP and variable-length GAs with homologous crossover. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)* Morgan Kaufmann, San Francisco, California, USA.

Poli, R., Rowe, J. E. & McPhee, N. F. (2001) Markov chain models for GP and variable-length GAs with homologous crossover. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)* Morgan Kaufmann, San Francisco, California, USA.

Poli, R., Rowe, J. E., Stephens, C. R. & Wright, A. H. (2002) Allele diffusion in linear genetic programming and variable-length genetic algorithms with subtree crossover. In *Proceedings of EuroGP 2002*.

Soule, T., Foster, J. A. & Dickinson, J. (1996) Code growth in genetic programming. In *Genetic Programming 1996: Proceedings of the First Annual Conference*, (Koza, J. R., Goldberg, D. E., Fogel, D. B. & Riolo, R. L., eds), pp. 215–223 MIT Press, Stanford University, CA, USA.

Spears, W. M. (2000) Limiting distributions for mutation and recombination. In *Proceedings of the Foundations of Genetic Algorithms Workshop (FOGA 6)*, (Spears, W. M. & Martin, W., eds), Charlottesville, VA, USA.

Stephens, C. R. (2001) Some exact results from a coarse grained formulation of genetic dynamics. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, (Spector, L., Goodman, E. D., Wu, A., Langdon, W. B., Voigt, H.-M., Gen, M., Sen, S., Dorigo, M., Pezeshk, S., Garzon, M. H. & Burke, E., eds), pp. 631–638 Morgan Kaufmann, San Francisco, California, USA.

Stephens, C. R. & Waelbroeck, H. (1997) Effective degrees of freedom in genetic algorithms and the block hypothesis. In *Proceedings of the Seventh International Conference on Genetic Algorithms (ICGA97)*, (Bäck, T., ed.), pp. 34–40 Morgan Kaufmann, East Lansing.

Stephens, C. R. & Waelbroeck, H. (1999) Schemata evolution and building blocks. *Evolutionary Computation,* **7** (2), 109–124.

Vose, M. D. & Wright, A. H. (2001) Form invariance and implicit parallelism. *Evolutionary Computation,* **9** (3), 355–370.

Iterative Refinement of Computational Circuits using Genetic Programming

Matthew J. Streeter
Genetic Programming, Inc.
Mountain View, California
mjs@tmolp.com

Martin A. Keane
Econometrics, Inc.
Chicago, Illinois
makeane@ix.netcom.com

John R. Koza
Stanford University
Stanford, California
koza@stanford.edu

Abstract

Previous work has shown that genetic programming is capable of creating analog electrical circuits whose output equals common mathematical functions, merely by specifying the desired mathematical function that is to be produced. This paper extends this work by generating computational circuits whose output is an approximation to the *error function* associated with an existing computational circuit (created by means of genetic programming or some other method). The output of the evolved circuit can then be added to the output of the existing circuit to produce a circuit that computes the desired function with greater accuracy. This process can be performed iteratively. We present a set of results showing the effectiveness of this approach over multiple iterations for generating squaring, square root, and cubing computational circuits. We also perform iterative refinement on a recently patented cubic signal generator circuit, obtaining a refined circuit that is 7.2 times more accurate than the original patented circuit. The iterative refinement process described herein can be viewed as a method for using previous knowledge (i.e. the existing circuit) to obtain an improved result.

1 INTRODUCTION

An analog electrical circuit whose output is a well-known mathematical function (e.g., square, square root) is called a *computational circuit*.

Analog computational circuits are especially useful when the mathematical function must be performed more rapidly than is possible with digital circuitry (e.g., for real-time signal processing at extremely high frequencies). Analog computational circuits are also useful when the need for a single mathematical function in an analog circuit does not warrant converting an analog signal into a digital signal (using an analog-to-digital converter), performing the mathematical function in the digital domain (requiring a general purpose digital processor consisting of millions of transistors), and then converting the result to the analog domain (using a digital-to-analog converter).

The design of computational circuits is exceedingly difficult even for seemingly mundane mathematical functions. Success usually relies on the clever exploitation of some aspect of the underlying device physics of the components (e.g., transistors) that is uniquely suited to the particular desired mathematical function. Because of this, the implementation of each different mathematical function typically requires an entirely different design approach (Gilbert 1968, Sheingold 1976, Babanezhad and Temes 1986).

The *topology* of a circuit involves specification of the total number of components in the circuit, the identity of each component (e.g., resistor, transistor), and the connections between each lead of each component. *Sizing* involves the specification of the values (typically numerical) of each component possessing a component value (e.g., resistors, capacitors). Genetic programming (Koza 1992; Koza and Rice 1992; 1994a, 1994b) is a technique for automatically creating computer programs to solve, or approximately solve, problems. Genetic programming is an extension of the genetic algorithm (Holland 1975). Genetic programming is capable of synthesizing the design of both the topology and component values (sizing) for a wide variety of analog electrical circuits from a high-level statement of the circuit's desired behavior and characteristics (Koza, Bennett, Andre, and Keane 1999; Koza, Bennett, Andre, Keane, and Brave 1999; Mydlowec and Koza 2000).

This paper extends previous work on synthesis of computational circuits using genetic programming by applying genetic programming in an iterative manner to obtain successively more accurate computational circuits. Specifically, we employ a multiple-run process in which each run involves generating a computational circuit that

produces as output an approximation to the *error function* (i.e. the difference between the target function and the circuit's actual output) of the best-of-run circuit from the previous run.

Section 2 of this paper presents a proof-of-principle experiment involving iterative refinement of rational polynomial approximations to the sine function. Section 3 describes automatic synthesis of electrical circuits by means of developmental genetic programming. Section 4 itemizes the preparatory steps required to apply genetic programming to the problem of synthesizing computational circuits. Section 5 describes our experiments involving iterative refinement of squaring, square root, and cubing computational circuits. Section 6 describes an experiment involving iterative refinement of a post-2000 patented cubic signal generator circuit. Section 7 is the conclusion.

2 ITERATIVE REFINEMENT OF NUMERICAL APPROXIMATIONS TO FUNCTIONS

Iterative refinement as described in this paper can be applied to other problem areas involving the synthesis of structures or entities that generate a specified target curve as output. In particular, it can be applied to the problem of generating numerical approximations to mathematical functions. As a proof-of-principle experiment that could be conducted using relatively modest computational resources, we evolved successively more accurate rational polynomial approximations to the function sin(x) over the interval $[0,\pi/2]$.

The function set for this problem consisted of the four arithmetic operators {+,*,-,%}. The terminal set consisted of the variable X and the random numeric terminal R. Fitness was defined as the sum of absolute error over 100 uniformly spaced points. We used a population size, M, of 100,000 and tournament selection with a tournament size of 10. The remaining control parameters are the same as those described in section 4.6 of this paper. Each run was conducted on a single Pentium workstation.

A first run (iteration 0) was conducted using the target function $\sin(x)$. The best-of-run individual from this first run produces an output $a_0(x)$, with an associated error function defined as $\sin(x)-a_0(x)$. We then perform a second run (iteration 1) using the target function $\sin(x)-a_0(x)$. This second run yields a best-of-run individual whose output $a_1(x)$ can be added to the result of the first run to produce a more accurate approximation $a_0(x)+a_1(x)$. This process was continued iteratively for a total of 6 runs including the initial run (with target function $\sin(x)$). The best-of-run rational polynomial from each of the runs was imported into the Maple symbolic mathematics package to calculate average error (using the integral of the error functions). Table 1 presents the average error of the best-of-run individuals for iteration N=0 through iteration N=5. R_{prev} indicates the ratio of improvement in average error over the previous iteration. R_{final} at the bottom of the table gives the ratio of improvement in accuracy of the best individual from all iterations over the best individual from the first iteration, i.e. the total ratio of improvement over all iterations.

Table 1 : Error of Rational Polynomial Approximations to sin(x), $[0,\pi/2]$ over Successive Iterations

N	ERROR	R_{prev}
0	4.554574e-6	—
1	3.483825e-8	130.73
2	2.093228e-8	1.6643
3	1.039170e-8	2.0143
4	6.302312e-9	1.6489
5	2.200276e-6	3.4912e-4

R_{final}: 772.65

As shown in Table 1, we obtain a significantly more accurate approximation for iterations 1 through 4 of our experiment. With each successive iteration, the error function being approximated becomes more complex (i.e. has more local extrema and sharper spikes). On iteration 5, we obtain a high-degree rational polynomial that, as evaluated under our genetic programming system over 100 fixed points, represents an improvement over the previous iteration. However, when imported into a symbolic mathematics package this approximation is revealed to have a large spike in between two of these 100 fixed points, giving it a substantially higher average error than the approximation produced by the previous iteration.

3 AUTOMATIC CIRCUIT SYNTHESIS USING DEVELOPMENTAL GENETIC PROGRAMMING

Both the topology and sizing of an electrical circuit can be created by genetic programming by means of a developmental process (Koza, Bennett, Andre, and Keane 1999; Koza, Bennett, Andre, Keane, and Brave 1999). This developmental process entails the execution of a circuit-constructing program tree that contains various component-creating, topology-modifying, and development-controlling functions. A simple initial circuit is the starting point of the developmental process for creating a fully developed electrical circuit. The initial circuit consists of an embryo and a test fixture. The embryo contains at least one modifiable wire. The test fixture is a fixed (hard-wired) substructure composed of nonmodifiable wire(s) and nonmodifiable electrical component(s). The test fixture provides access to the

circuit's external input(s) and permits probing of the circuit's output. All development originates from the modifiable wires. The execution of the component-creating, topology-modifying, and development-controlling functions in the program tree transforms the initial circuit into a fully developed circuit. We use the methods described by Koza, Bennett, Andre and Keane (1999) in combination with the four technical modifications described by Streeter, Keane, and Koza (2002).

4 PREPARATORY STEPS

4.1 INITIAL CIRCUIT

A one-input, one-output initial circuit with one modifiable wire was used for all problems described in this paper with the exception of the problem involving the patented cubic signal generator circuit (described in section 6). The initial circuit has an incoming signal source, a 1 μOhm source resistor, a voltage probe point VOUT, and a 1 gigaOhm load resistor. The topology of the circuit is the same as that shown in figure 30.1 of Koza, Bennett, Andre, and Keane 1999.

4.2 PROGRAM ARCHITECTURE

The circuit-constructing program tree has one result-producing branch for each modifiable wire in the embryo of the initial circuit. Thus, each program tree has one result-producing branch for each of the problems described in this paper. Automatically defined functions were not used.

4.3 TERMINAL SET

The value of each component in a circuit possessing a parameter (e.g., resistors) is established by an argument of its component-creating function. The argument is a numerical terminal whose value can be perturbed by a special genetic operation for mutation of constants. Aside from the numerical constants, the terminal set for each result-producing branch is

T_{rpb} = {END, SAFE_CUT, B_C_E ... E_C_B, BIFURCATE_POSITIVE, BIFURCATE_NEGATIVE, Q2N3906, Q2N3904, UP_OR_LEFT, DOWN_OR_RIGHT}.

These terminals are each described in detail in Koza, Bennett, Andre, and Keane 1999 and Streeter, Keane, and Koza 2002. Briefly, the END terminal is a development-controlling function that ends the developmental process for a particular path through the circuit-constructing program tree. SAFE_CUT is a topology-modifying function that deletes a modifiable wire or component from the developing circuit while preserving circuit validity. The B_C_E through E_C_B terminals specify which of six possible permutations to use for the three leads of a transistor when inserting a transistor into a circuit. The BIFUCATE_POSITIVE and BIFURCATE_NEGATIVE terminals specify which end of the modifiable wire to bifurcate when inserting a transistor. The Q2N3906 and Q2N3904 terminals specify which of two available transistor models to use when inserting a transistor. The DOWN_OR_RIGHT and UP_OR_LEFT terminals specify the two possible pairs of directions in which parallel division can be performed.

4.4 FUNCTION SET

The function set for each result-producing branch is

F_{rpb} = {R, Q, SERIES, PARALLEL, NODE, TWO-LEAD, TWO_GROUND, THREE_GROUND}.

See Koza, Bennett, Andre, and Keane 1999 and Streeter, Keane, and Koza 2002 for a detailed description of each of these functions. Briefly, the R function is a component-creating function that creates a resistor with a resistance specified as an argument to the function. A function that creates a two-leaded component (with R here being the only choice) can be used as an argument to the TWO-LEAD function, which inserts two-leaded components into the circuit. The Q function inserts transistors into a developing circuit. The SERIES and PARALLEL functions modify the topology of the developing circuit by performing a series or parallel division (respectively). The NODE function is used to connect distant points in a circuit. The TWO_GROUND and THREE_GROUND functions each create a via to ground.

4.5 FITNESS MEASURE

The evaluation of each individual circuit-constructing program tree in the population begins with its execution. The execution progressively applies the component-creating, topology-modifying, and development-controlling functions in the program tree to the embryo of the initial circuit (and to intermediate circuits during the developmental process), thereby eventually yielding a fully developed circuit. A netlist is then created that identifies each component of the fully developed circuit, the nodes to which each component is connected, and the numerical value of each component. The netlist becomes the input to our modified version of the SPICE (Simulation Program with Integrated Circuit Emphasis) simulation program (Quarles, Newton, Pederson, and Sangiovanni-Vincentelli 1994). SPICE determines the circuit's behavior in terms of the output voltage VOUT in the time domain. Each individual circuit in the population is exposed to two time-domain signals for a (simulated) duration of either one or one hundred seconds. The first time domain signal is a straight line rising from 0 to 1 volt over a period of one second. The second time domain signal is a straight line falling from 1 volt to 0 volts over 100 seconds. Each time-domain simulation is run for 100 time steps.

Absolute error is defined as the sum, over each time step, of the absolute difference between the output of the circuit and the value of the target function (which varies from problem to problem). Overall fitness consists of a

weighted sum of absolute error. The error of each point is weighted by a factor of 10 if it is not a hit, and a factor of 1 otherwise. A hit is defined as an output that is within 1% of the desired value.

4.6 CONTROL PARAMETERS

For the computational circuit problems described in this paper, the population size, *M*, was 20,000. A (generous) maximum size of 500 points (i.e., total number of functions and terminals) was established for the result-producing branch. The percentages of the genetic operations are 60% one-offspring crossover on internal points of the program tree other than numerical constant terminals, 10% one-offspring crossover on points of the program tree other than numerical constant terminals, 1% mutation on points of the program tree other than numerical constant terminals, 20% perturbation on numerical constant terminals, and 9% reproduction. The other parameters are the same default values that we have used previously on a broad range of problems (Koza, Bennett, Andre, Keane 1999).

4.7 TERMINATION

Each run (performed as part of a multi-run process) was allowed to perform 100 generations before being terminated.

4.8 PARALLEL IMPLEMENTATION

Each of the problems described in this paper was run on a home-built Beowulf-style (Sterling, Salmon, Becker, and Savarese 1999; Bennett, Koza, Shipman, and Stiffelman 1999) parallel cluster computer system consisting of 20 350 MHz Pentium II processors (each accompanied by 64 megabytes of RAM). These 20 processors were isolated into a logically separate cluster for the purpose of these experiments. They normally act as part of a larger 1,000 processor cluster which we apply to more difficult problems involving genetic programming. The system has a 350 MHz Pentium II computer as host.

The processing nodes are connected with a 100 megabit-per-second Ethernet. The processing nodes and the host use the Linux operating system. The distributed genetic algorithm with unsynchronized generations and semi-isolated subpopulations was used with a subpopulation size of Q = 1,000 at each of D = 20 demes. As each processor (asynchronously) completes a generation, four boatloads of emigrants from each subpopulation are dispatched to each of the four toroidally adjacent processors. Emigrants are selected randomly and the migration rate is 5% (10% if the adjacent node is in the same physical box).

5 ITERATIVE REFINEMENT OF EVOLVED COMPUTATIONAL CIRCUITS

5.1 RESULTS

We applied our iterative refinement process to the generation of squaring, square root, and cubing computational circuits. In each case, a first run of genetic programming was conducted to create a circuit which output the desired function with some degree of accuracy. A second run was then performed in which the target function was taken as the error function associated with the best-of-run circuit from the previous run. This process was continued for successive iterations until a run produced an improvement in accuracy over the previous iteration of 5% or less. Tables 2, 3, and 4 present the results of these experiments for squaring, square root, and cubing computational circuits, respectively. Figures 1, 3, and 5 present the output curves for the circuits produced by successive iterations for the squaring, square root, and cubing functions, respectively. Figures 2, 4, and 6 present the error curves over successive iterations for the squaring, square root, and cubing circuits, respectively.

Table 2 : Error of Squaring Computational Circuits over Successive Iterations

N	ERROR (mV)	R_{prev}
0	46.84	—
1	5.836	8.0260
2	5.5635	1.0490

R_{final}: 8.4193

Table 3 : Error of Square Root Computational Circuits over Successive Iterations

N	ERROR (mV)	R_{prev}
0	11.835	—
1	3.4445	3.4359
2	3.398	1.0137

R_{final}: 3.4830

Table 4 : Error of Cubing Computational Circuits over Successive Iterations

N	ERROR (mV)	R_{prev}
0	22.312	—
1	20.198	1.1047
2	17.510	1.1535
3	17.061	1.0263

$\mathbf{R_{final}}$: 1.3078

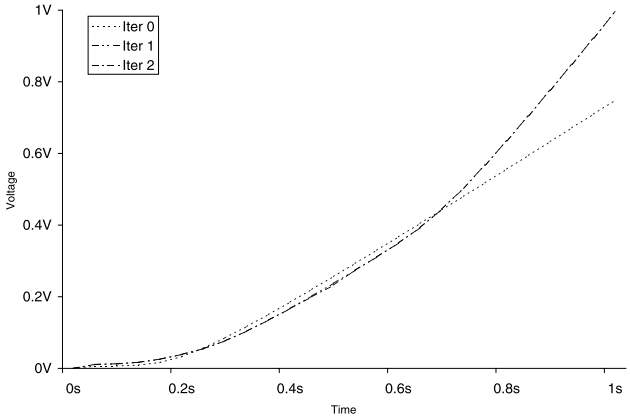


Figure 1: Output of Squaring Computational Circuits over Successive Iterations

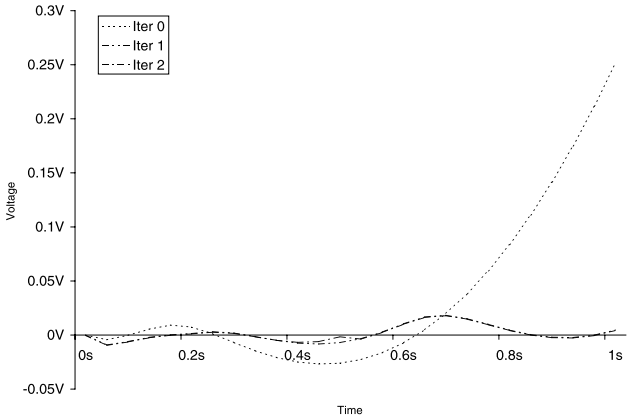


Figure 2: Error of Squaring Computational Circuits over Successive Iterations

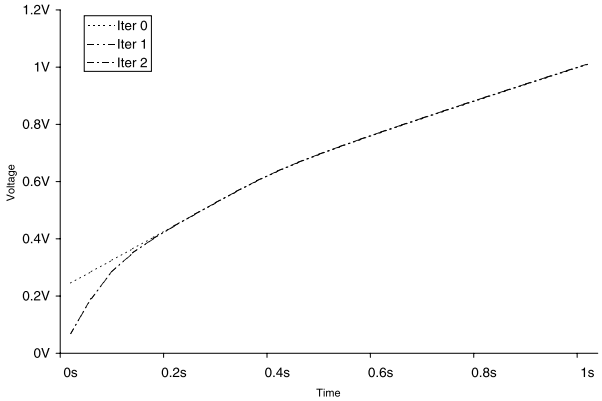


Figure 3: Output of Square Root Computational Circuits over Successive Iterations

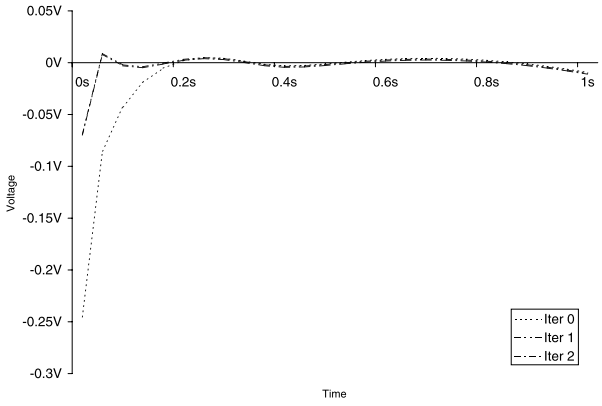


Figure 4: Error of Square Root Computational Circuits over Successive Iterations

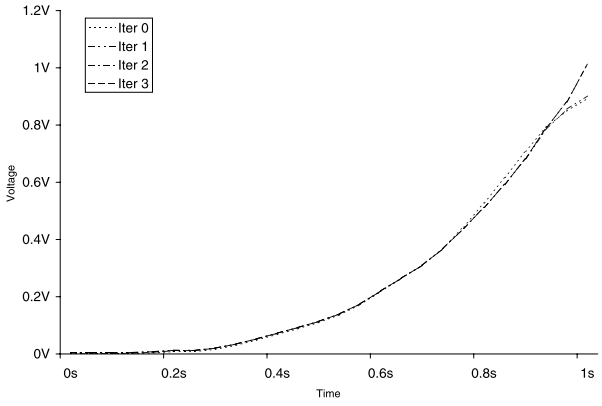


Figure 5: Output of Cubing Computational Circuits over Successive Iterations

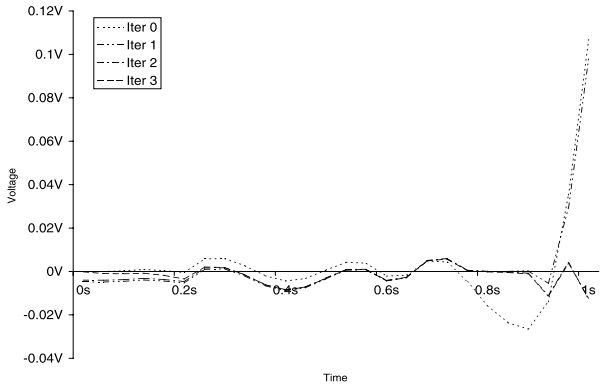


Figure 6: Error of Cubing Computational Circuits over Successive Iterations

5.2 EFFICIENCY OF EVOLVING COMPUTATIONAL CIRCUITS USING ITERATIVE REFINEMENT

It is not necessarily (and almost undoubtedly not) the case that the experiments described above produce their results with the most efficient possible use of available computational resources. Since our multi-run process involved 6 separate GP runs, the same computational resources could be expended by executing (for example) a single run for 6 times as long or a single run using 6 times the population. We have no evidence to show that either of these approaches, or some other approach which used the same amount of computational resources as our experiments, would not produce better results than the ones given here. However, our purpose here is to establish that genetic programming is capable of creating computational circuits whose output is an approximation to the error function of another computational circuit, and that the iterative process used in these experiments is indeed able to create successively more accurate computational circuits.

Also note that the results presented in this section and in section 2 in no way imply that the error functions being targeted in each successive iteration are somehow more amenable to approximation by genetic programming than are the original target functions (i.e. sqrt(x), x^2, and x^3). Rather, it is the fact that genetic programming is able to produce circuits whose outputs approximate the error functions with any degree of accuracy at all that allows successively more accurate circuits to be obtained.

6 REFINEMENT OF A POST-2000 PATENTED CIRCUIT

U.S. patent 6,160,427 covers a low-voltage cubic signal generator circuit that produces an output current approximately equal to the cube of its input current (Cipriani and Takeshian, 2000). In a previous paper, we successfully applied genetic programming to the synthesis of a computational circuit that duplicates the functionality of this patented circuit. Specifically, we evolved a circuit that meets the low voltage criteria defined in the patent and has an average error that is 59% of that of the patented circuit over the four fitness cases on which it was evaluated (Streeter, Keane, and Koza 2002).

We now apply our iterative refinement process to this patented circuit. The preparatory steps for the runs performed in this experiment differ from the preparatory steps given in section 4 in four minor ways. First, we force the evolved circuit to conform to the low voltage specification in the patent by including only a 2V power supply in the function set, rather than the 15V and –15V supplies we had included in previous run. Second, we desire that this circuit make full use of the voltage range provided by the power supply, i.e. the circuit's output should range from 0V to 2V. This means that the input voltage should have a minimum value of 0V and a maximum voltage equal to the cube root of 2. The two time-domain curves described in section 4.5 are modified accordingly. Third, for this problem we use only a minimal embryo consisting of a single modifiable wire initially not connected to the test fixture, in contrast to the embryo consisting of a modifiable wire connected to source and load resistors as described in section 4.1. We generally use this minimal embryo when dealing with a test fixture that has multiple inputs or outputs. In this case there is one input (representing an input current flowing down from a 2V supply) and two outputs (representing the high and low points at which the output current is to be probed). Additional functions are included in the function set that allow connections to be made to the single input and the two outputs. For more information on these functions see Koza, Bennett, Andre, Keane 1999. Finally, each of the runs described in this section was manually monitored and terminated when it appeared to have reached a plateau.

When building upon a patented circuit, the first iteration (iteration 0) of our iterative process does not involve a GP run, but rather involves building and simulating the patented circuit based on the schematics given in the patent document. Our first GP run (iteration 1) involves the synthesis of a computational circuit whose output approximates the error function associated with the patented circuit. Our second GP run (iteration 2) involves the synthesis of a circuit whose output approximates the error function of the best-of-run circuit from our first GP run. The results of these runs are presented in Table 5.

Table 5 : Error of Low-Voltage Cubing Circuits over Successive Iterations

N	ERROR (mV)	R_{prev}
0	7.128	—
1	0.9873	7.2197
2	0.9236	1.0690

R_{final}: 7.2197 (see below)

The first iteration of our experiment produced a circuit whose average error was 7.2 times better than that of the patented circuit (over our two fitness cases). This circuit was created after 147 generations. The circuit was tested on a variety of unseen fitness cases, and outperformed the patented circuit on those fitness cases as well. The second iteration of our experiment produced a circuit that was approximately 1.07 times better than the circuit of the previous iteration. This circuit was created after 80 generations. However, examination of the output of this circuit reveals that it contains a number of sharp spikes which indicate instability in the circuit's output. This circuit generally does not perform better that the circuit of the previous iteration when tested on unseen fitness cases.

The output curves for the circuits produced in iterations 0 and 1 of our experiment are given in Figure 7. The error curves for these two circuits are given in Figure 8.

Figure 9 presents the refined computational circuit obtained by adding the output of the patented cubic signal generator circuit to the output of the evolved circuit. In this figure there are two rectangles drawn using dotted lines. The top rectangle encloses the circuitry from the patented cubic signal generator, while the bottom rectangle encloses the evolved circuitry. Components which are outside both rectangles are part of the test fixture for the problem. The two outputs labeled `VITER0` (for the output of the patented circuit) and `VITER1` (for the output of the evolved circuit) are passed through a voltage addition block to produce the final output (`VFINAL`) of the refined circuit.

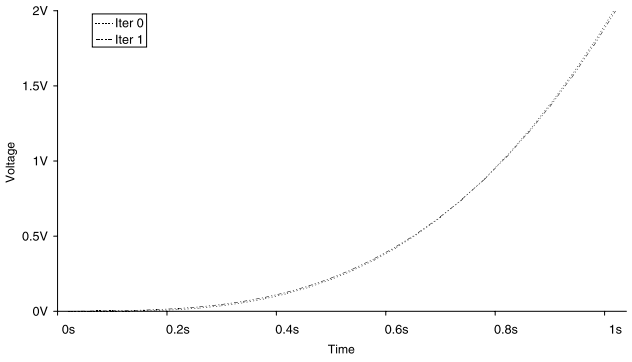


Figure 7: Output of Low-Voltage Cubing Circuits over Successive Iterations

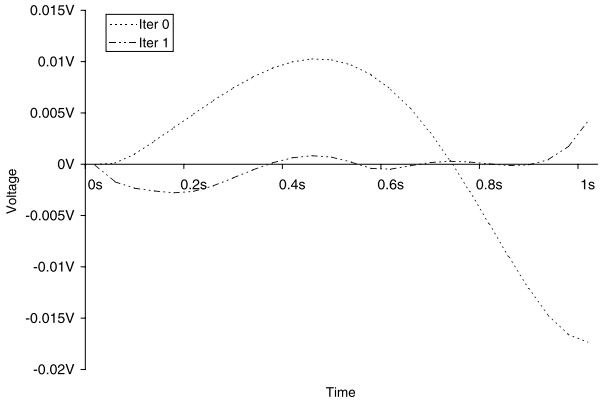


Figure 8: Error of Low-Voltage Cubing Circuits over Successive Iterations

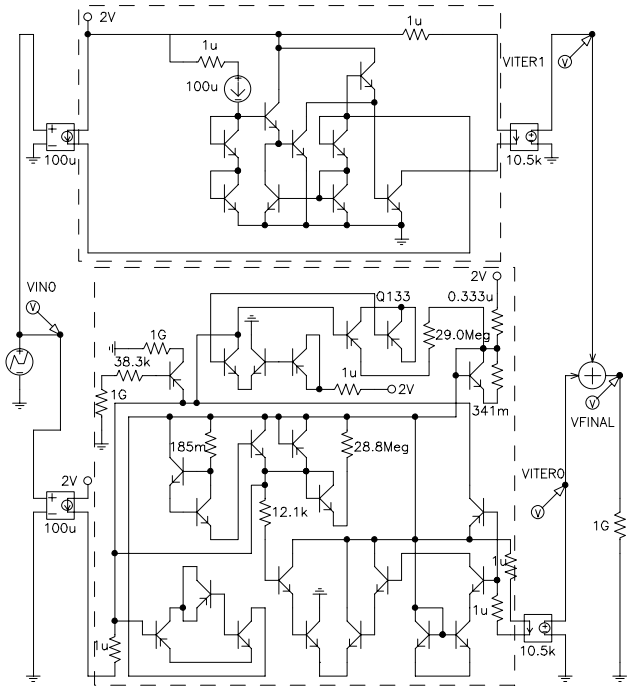


Figure 9: Refined Low-Voltage Cubic Signal Generator Circuit

As previously mentioned, computational circuits are valuable in that they are able to produce the output of complex mathematical functions at near-instantaneous analog speed. However, no human-designed computational circuit (or circuit designed by any other means) is likely to produce the desired output with perfect accuracy. For this reason, an error correction process is desireable. Genetic programming has been shown in this paper to provide such a process. Since the shape of the error functions associated with computational circuits are rather arbitrary and do not correspond to simple mathematical functions, it would be difficult if not impossible for an analog engineer to generate a circuit that produced these functions. The synthesis of computational circuits that produce these error functions as output is therefore an especially suitable application of genetic programming.

7 CONCLUSIONS

We have shown that an iterative refinement process can be applied both to the generation of rational polynomial approximations to functions and to the synthesis of computational circuits using genetic programming. In particular, we have applied this process to the synthesis of squaring, square root, and cubing computational circuits, obtaining an increase in accuracy with each successive iteration of refinement. We further applied this iterative process to a recently patented low-voltage cubic signal generator circuit, and achieved an improvement in accuracy of a factor of 7.2 over the patented circuit. We conclude that the approach described in this paper provides an effective way to generate high-accuracy computational circuits.

References

Babanezhad, J. N. and Temes, G. C. 1986. Analog MOS Computational Circuits. *Proceedings of the IEEE Circuits and System International Symposium*. Piscataway, NJ: IEEE Press. Pages 1156–1160.

Bennett, Forrest H III, Koza, John R., Shipman, James, and Stiffelman, Oscar. 1999. Building a parallel computer system for $18,000 that performs a half peta-flop per day. In Banzhaf, Wolfgang, Daida, Jason, Eiben, A. E., Garzon, Max H., Honavar, Vasant, Jakiela, Mark, and Smith, Robert E. (editors). 1999. *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference, July 13-17, 1999, Orlando, Florida USA*. San Francisco, CA: Morgan Kaufmann. 1484 - 1490.

Cipriani, Stefano and Takeshian, Anthony A. 2000. *Compact cubic function generator*. U. S. patent 6,160,427. Filed September 4, 1998. Issued December 12, 2000.

Gilbert, Barrie. 1968. A precise four-quadrant multiplier with subnanosecond response. *IEEE Journal of Solid-State Circuits*. Volume SC-3. Number 4. December 1968. Pages 365–373.

Holland, John H. 1975. *Adaptation in Natural and Artificial Systems*. 2nd. Ed. Cambridge, MA: MIT Press.

Koza, John R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, MA: MIT Press.

Koza, John R. 1994a. *Genetic Programming II: Automatic Discovery of Reusable Programs.* Cambridge, MA: MIT Press.

Koza, John R. 1994b. *Genetic Programming II Videotape: The Next Generation*. Cambridge, MA: MIT Press.

Koza, John R., Bennett III, Forrest H, Andre, David, and Keane, Martin A. 1999. *Genetic Programming III: Darwinian Invention and Problem Solving*. San Francisco, CA: Morgan Kaufmann.

Koza, John R., Bennett III, Forrest H, Andre, David, Keane, Martin A., and Brave, Scott. 1999. *Genetic Programming III Videotape*: *Human-Competitive Machine Intelligence*. San Francisco, CA: Morgan Kaufmann.

Koza, John R., and Rice, James P. 1992. *Genetic Programming: The Movie*. Cambridge, MA: MIT Press.

Mydlowec, William and Koza, John R. 2000. Use of Time-Domain Simulations in Automatic Synthesis of Computational Circuits using Genetic Programming. In Whitley, Darrell (editor). 2000. *Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference*. Las Vegas, NV: Morgan Kaufmann. 187-197.

Quarles, Thomas, Newton, A. R., Pederson, D. O., and Sangiovanni-Vincentelli, A. 1994. *SPICE 3 Version 3F5 User's Manual*. Department of Electrical Engineering and Computer Science, University of California. Berkeley, CA. March 1994.

Sheingold, Daniel H. (editor). 1976. *Nonlinear Circuits Handbook*. Norwood, MA: Analog Devices, Inc.

Sterling, Thomas L., Salmon, John, Becker, Donald J., and Savarese, Daniel F. 1999. *How to Build a Beowulf: A Guide to Implementation and Application of PC Clusters*. Cambridge, MA: MIT Press.

Streeter, Matthew J., Keane, Martin A., and Koza, John R. 2002. Routine Duplication of Post-2000 Patented Inventions by Means of Genetic Programming. In Lutton, Evelyn, Foster, James A., Miller, Julian, Ryan, Conor and Tettamanzi, Andrea. *Proceedings of EuroGP'2002, April 3-5, 2002, Kinsale, Ireland*. Springer-Verlag. 26-36.