Covariant Parsimony Pressure for Genetic Programming

Riccardo Poli
Department of Computing and Electronic Systems
University of Essex, UK
rpoli@essex.ac.uk

Nicholas F. McPhee
Division of Science and Mathematics
University of Minnesota, Morris, USA
mcphee@morris.umn.edu

Technical Report CES-480
ISSN: 1744-8050
January 2008

Abstract

The parsimony pressure method is perhaps the simplest and most frequently used method to control bloat in genetic programming. In this paper we first reconsider the size evolution equation for genetic programming developed in [24] and rewrite it in a form that shows its direct relationship to Price's theorem. We then use this new formulation to derive theoretical results that show how to practically and optimally set the parsimony coefficient dynamically during a run so as to achieve complete control over the growth of the programs in a population. Experimental results confirm the effectiveness of the method, as we are able to tightly control the average program size under a variety of conditions. These include such unusual cases as dynamically varying target sizes such that the mean program size is allowed to grow during some phases of a run, while being forced to shrink in others.

1 Introduction

Starting in the early '90s researchers began to notice that in addition to progressively increasing their mean and best fitness, GP populations also exhibited certain other dynamics. In particular, it was often observed that, while the average size (number of nodes) of the programs in a population was initially fairly static (if noisy), at some point the average

program size would start growing at a rapid pace. Typically this increase in program size was not accompanied by any corresponding increase in fitness.

This phenomena, which is known as *bloat*, has been the subject of intense study in GP, both because of its initially surprising nature, and because of its significant practical effects. (Large programs are computationally expensive to evolve and later use, can be hard to interpret, and may exhibit poor generalisation.) Over the years, many theories have been proposed to explain various aspects of bloat, and while great strides have been made, we still lack a single unifying theory to explain the broad range of empirical observations. We review the key theoretical results on bloat in Section 2, with special emphasis on the size evolution equation [24] which forms the basis for the new approach presented here.

While the jury remains out on the causes of bloat, practitioners have still had to face the reality of combating bloat in their runs. Consequently, a variety of practical techniques have been proposed to counteract bloat; we review these in Section 3. We will particularly focus on the *parsimony pressure method* [12, 33, 34, 35], which is perhaps the simplest and most frequently used method to control bloat in genetic programming. This method effectively treats the minimisation of size as a soft constraint and attempts to enforce this constraint using the penalty method, i.e., by decreasing the fitness of programs by an amount that depends on their size. The penalty is typically simply proportional to program size. The intensity with which bloat is controlled is, therefore, determined by one parameter called the *parsimony coefficient*. The value of this coefficient is very important: too small a value and runs will still bloat wildly; too large a value and GP will take the minimisation of size as its main target and will almost ignore fitness, thus converging towards extremely small but useless programs. Unfortunately, however, the "correct" values of this coefficient are highly dependent on particulars such as the problem being solved, the choice of functions and terminals, and various parameter settings, and very little theory has been put forward to aid in setting the parsimony coefficient in a principled manner. This has remained an outstanding problem in GP for over a decade, with users being forced to proceed by trial and error.

This paper presents a simple, effective, and theoretically grounded solution to this problem. In Section 4, we reconsider the size evolution equation for GP developed in [24], rewriting it in a form that shows its direct relationship to Price's theorem [25, 16]. We then use this new formulation to derive theoretical results that tells us how to practically and optimally set the parsimony coefficient dynamically during a run so as to achieve very tight control over the average size of the programs in a population. We test our theory in Section 5 where we report extensive empirical results, showing how accurately the method controls program size in a variety of conditions. We then conclude in Section 6.

2 Bloat in Theory

As mentioned above, there are several theories of bloat. For example, the *replication accuracy theory* [19] states that the success of a GP individual depends on its ability to have offspring that are functionally similar to the parent. As a consequence, GP evolves towards

(bloated) representations that increase replication accuracy. The *removal bias theory* [32] observes that inactive code in a GP tree (code that is not executed, or is executed but its output is then discarded) tends to be low in the tree, residing therefore in smaller-than-average-size subtrees. Crossover events excising inactive subtrees produce offspring with the same fitness as their parents. On average the inserted subtree is bigger than the excised one, so such offspring are bigger than average while retaining the fitness of their parent, leading ultimately to growth in the average program size. Another important theory, the *nature of program search spaces theory* [15, 17], predicts that above a certain size, the distribution of fitnesses does not vary with size. Since there are more long programs, the number of long programs of a given fitness is greater than the number of short programs of the same fitness. Over time GP samples longer and longer programs simply because there are more of them.

The explanations for bloat mentioned above are largely qualitative. There have, however, been some efforts to mathematically formalise and verify these theories. For example, Banzhaf and Langdon [3] defined a executable model where only the fitness, the size of active code and the size of inactive code were represented (i.e., there was no representation of program structures). Fitnesses of individuals were drawn from a bell-shaped distribution, while active and inactive code lengths were modified by a size-unbiased mutation operator. Various interesting effects were reported which are very similar to corresponding effects found in GP runs. Rosca proposed a similar, but slightly more sophisticated model which also included an analogue of crossover [27]. This provided further interesting evidence. A strength of these types of models is their simplicity. A weakness is that they suppress or remove many details of the representation and operators typically used in GP. This makes it difficult to verify if all the phenomena observed in the model have analogues in GP runs, and if all important behaviours of GP in relation to bloat are captured by the model.

In [21, 24], a *size evolution equation* for genetic programming was developed, which is an exact formalisation of the dynamics of average program size:

$$E[\mu(t+1)] = \sum_l S(G_l)p(G_l,t), \tag{1}$$

Here $\mu(t+1)$ is the mean size of the programs in the population at generation $t+1$, E is the expectation operator, G_l is the set of all programs of a particular shape l, $S(G_l)$ is the size of programs in the set G_l (i.e., programs having the shape l), and $p(G_l,t)$ is the probability of selecting programs from G_l (i.e., of shape l) from the population in generation t. This can be rewritten in terms of the expected change in average program size as:

$$E[\mu(t+1) - \mu(t)] = \sum_l S(G_l)(p(G_l,t) - \Phi(G_l,t)), \tag{2}$$

where $\Phi(G_l,t)$ is the proportion of programs of shape G_l in the current generation. Both equations apply to a GP system with selection and any form of symmetric subtree crossover.[1]

[1] In a symmetric operator the probability of selecting particular crossover points in the parents does not depend on the order in which the parents are drawn from the population.

Note that Equations (1) and (2) do not directly explain bloat. They are, however, important because they constrain what can and cannot happen size-wise in GP populations. So, any explanation for bloat (including the theories summarised in this section), if correct, has to agree with these results. In particular, Equation (1) predicts that, for symmetric subtree-swapping crossover operators, the mean program size evolves as if selection only was acting on the population. This means that if there is a variation in mean size (bloat, for example) it must be the result of some form of positive or negative selective pressure on some or all of the shapes G_l. Equation (2) shows that there can be bloat only if the selection probability $p(G_l, t)$ is different from the proportion $\Phi(G_l, t)$ for at least some l. In particular, for bloat to happen there will have to be some short G_l's for which $p(G_l, t) < \Phi(G_l, t)$ and also some longer G_l's for which $p(G_l, t) > \Phi(G_l, t)$ (at least on average). As we will see later, Equations (1) and (2) are the starting point for the work reported in this paper.

We conclude this review with a recent explanation for bloat called the *crossover bias theory* [23, 5] which is based in significant part and is consistent with the size evolution equation above. On average, each application of subtree crossover removes as much genetic material as it inserts. So, crossover on its own does not produce growth or shrinkage. However, while the *mean* program size is unaffected, *higher moments* of the distribution are. In particular, crossover pushes the population towards a particular distribution of program sizes (a Lagrange distribution of the second kind), where small programs have a much higher frequency than longer ones. For example, crossover generates a very high proportion of single-node individuals. In virtually all problems of practical interest, very small programs have no chance of solving the problem. As a result, programs of above average length have a selective advantage over programs of below average length. Consequently, the mean program size increases.

3 Bloat/Anti-bloat in Practice

Numerous empirical techniques have been proposed to control bloat [17, 31]. Among the most recent are *size fair crossover* and *size fair mutation* [14, 4]. These work by constraining the choices made during the execution of a genetic operation so as to actively prevent growth. In size-fair crossover, for example, the crossover point in the first parent is selected randomly, as in standard crossover. Then the size of the subtree to be excised is calculated. This is used to constrain the choice of the second crossover point so as to guarantee that the subtree chosen from the second parent will not be "unfairly" big. Another recent technique, the *Tarpeian method* [22], controls bloat by acting directly on the selection probabilities in Equation 2. This is done by setting the fitness of randomly chosen longer than average programs to 0. Recent methods also include the use of *multi-objective optimisation* (with two objectives: fitness and size) to control bloat. For example, [6] used a modified selection based on the Pareto criterion to reduce code growth without significant loss of solution accuracy.

Older methods include several *mutation operators* that may help control the average tree

size in the population while still introducing new genetic material. [10] proposes a mutation operator which prevents the offspring's depth being more then 15% larger than its parent. [13] proposes two mutation operators in which the new random subtree is on average the same size as the code it replaces. In *Hoist mutation* [11] the new subtree is selected from the subtree being removed from the parent, guaranteeing that the new program will be smaller than its parent. *Shrink mutation* [2] is a special case of subtree mutation where the randomly chosen subtree is replaced by a randomly chosen terminal. [20] provides theoretical analysis and empirical evidence that combinations of subtree crossover and subtree mutation operators can control bloat in linear GP systems.

None of the methods mentioned above, however, has gained as much widespread acceptance as the *parsimony pressure method* [12, 33, 34, 35]. The method works as follows. Let $f(x)$ be the fitness of program x. When the parsimony pressure is applied we define and use a new fitness function

$$f_p(x) = f(x) - c\ell(x) \tag{3}$$

where $\ell(x)$ is the size of program x and c is a constant known as the *parsimony coefficient.*[2] [34] showed the benefits of adaptively adjusting the coefficient c at each generation in experiments on the evolution of Sigma-Pi neural networks with GP , but most implementations and results in the literature actually keep c constant. As we will see in Section 4, however, a dynamic c is in fact essential to obtain full control of bloat.

The parsimony pressure method can be seen as a way to address the generalisation-accuracy tradeoff common in machine learning [34, 29]. There are also connections between this method and the Minimum Description Length (MDL) principle used to control bloat in [8, 9, 7]. The MDL approach uses a fitness function which combines program complexity (expressed as the number of bits necessary to encode the program's tree) and classification error (expressed as the number of bits necessary to encode the errors on all fitness cases). Rosca also linked the parsimony pressure method to his approximate evolution equations for rooted-tree schemata [29, 26, 28, 30].

Naturally, controlling bloat while at the same time maximising fitness turns the evolution of programs into either a multi-objective optimisation problem or, at least, into a constrained optimisation problem. Thus, as mentioned in Section 1, we should expect (and numerous results in the literature show this) that excessively aggressive methods to control bloat may lead to poor performance (in terms of ability to solve the problem at hand) of the evolved programs. The parsimony pressure method is not immune from this risk. So, although good control of bloat can be obtained with a careful choice of the parsimony coefficient, the choice of such a coefficient is an important but delicate matter. To date, however, trial and error remains the only general method for setting the parsimony coefficient. Furthermore, with a constant c the method can only achieve *partial control* over the dynamics of the average program size over time.

[2]Naturally, while f_p is used to guide evolution, one needs to still use the original fitness function f to recognise solutions and stop runs.

In this paper we aim to change all that, theoretically deriving and testing an easy and practical modification of the parsimony pressure technique which provides *extremely tight* control over the dynamics of the mean program size.

4 Size evolution, Price and Optimal Parsimony Pressure

In this section we show the relationship between the size evolution equation and Price's Theorem [25]. We also show how to use this new form of the size evolution equation to solve for dynamic parsimony coefficients that will allow for various types of control of the average program size (e.g., Equations (14), (15), (19), and (20)). While some of these may look daunting at first, they are in fact straightforward to add to most GP systems and (as is shown in Section 5) can provide exceptionally tight control over the average population size.

Let us start by considering Equation (1) again. With trivial manipulations it can be rewritten in terms of length-classes, rather than tree shapes, obtaining

$$E[\mu(t+1)] = \sum_{\ell} \ell p(\ell, t) \tag{4}$$

where the index ℓ ranges over all program sizes, and

$$p(\ell, t) = \sum_{l:S(G_l)=\ell} p(G_l, t). \tag{5}$$

Similarly, we can then rewrite Equation (2) as

$$E[\Delta\mu] = E[\mu(t+1) - \mu(t)] = \sum_{\ell} \ell \left(p(\ell, t) - \Phi(\ell, t)\right), \tag{6}$$

where $\Phi(\ell, t) = \sum_{l:S(G_l)=\ell} \Phi(G_l, t)$.

We now restrict our attention to fitness proportionate selection. In this case

$$p(\ell, t) = \Phi(\ell, t) \frac{f(\ell, t)}{\bar{f}(t)}, \tag{7}$$

where $f(\ell, t)$ is the average fitness of the programs of size ℓ and $\bar{f}(t)$ is the average fitness of the programs in the population, both computed at generation t. Then from Equation (6) we obtain

$$\begin{aligned} E[\Delta\mu] &= \sum_{\ell} \ell \left(\Phi(\ell, t) \frac{f(\ell, t)}{\bar{f}(t)} - \Phi(\ell, t) \right) \\ &= \frac{1}{\bar{f}(t)} \sum_{\ell} \ell (f(\ell, t) - \bar{f}(t)) \Phi(\ell, t) \end{aligned}$$

$$= \frac{1}{\bar{f}(t)}\Big(\underbrace{\sum_{\ell}(\ell-\mu(t))(f(\ell,t)-\bar{f}(t))\Phi(\ell,t)}_{=\ \mathrm{Cov}(\ell,f)\ \text{ by definition}} + \mu(t)\underbrace{\sum_{\ell}(f(\ell,t)-\bar{f}(t))\Phi(\ell,t)}_{=\ 0\ \text{by definition of}\ \bar{f}(t)}\Big),$$

where $\mu(t) = \sum_\ell \ell\,\Phi(\ell,t)$ is the current average program size. So,

$$E[\Delta\mu] = \frac{\mathrm{Cov}(\ell,f)}{\bar{f}(t)}. \tag{8}$$

This result is important because it shows that Equation (6), our coarse grained version of Equation (2), is in fact a form of Price's theorem (see [25, 16, 1] for a detailed review). While Price's theorem is generally applicable to "inheritable features" in an evolving system, only informal arguments have so far been made conjecturing that size might be one such features [16]. Our result, instead, *proves* the conjecture.

We are now in a position to more clearly see the effects of parsimony pressure and, more generally, of any form of program-size control based on the following generalisation of Equation (3):

$$f_p(x,t) = f(x) - g(\ell(x),t) \tag{9}$$

where g is a function of program size, $\ell(x)$, and generation, t. To achieve this we consider Equation (8) when the fitness function $f(x)$ is replaced by $f_p(x,t)$. We obtain

$$E[\Delta\mu] = \frac{\mathrm{Cov}(\ell,f_p)}{\bar{f}_p} \tag{10}$$

$$= \frac{\mathrm{Cov}(\ell,f-g)}{\bar{f}-\bar{g}} \tag{11}$$

$$= \frac{\mathrm{Cov}(\ell,f)-\mathrm{Cov}(\ell,g)}{\bar{f}-\bar{g}} \tag{12}$$

where we omitted t for brevity. So, absence of growth (and bloat), $E[\Delta\mu] = 0$, is obtained if

$$\mathrm{Cov}(\ell,g) = \mathrm{Cov}(\ell,f). \tag{13}$$

In many conditions, this equation makes it possible to determine the penalty function g to use to control the program size dynamics in GP runs.

As an example, let us consider the case $g(\ell(x),t) = c(t)\ell(x)$ where $c(t)$ is a function of the generation number t. Here we have that

$$\mathrm{Cov}(\ell,g) = c(t)\,\mathrm{Cov}(\ell,\ell) = c(t)\,\mathrm{Var}(\ell)$$

Substituting this into Equation (13) and solving for $c(t)$ one finds that, in order to completely remove growth (or shrinking) from a run, one needs to set

$$c(t) = \frac{\mathrm{Cov}(\ell, f)}{\mathrm{Var}(\ell)}. \tag{14}$$

Note that c is a function of t because both numerator and denominator can change from generation to generation.

Let us now consider the more general case $g(\ell(x), t) = c(t)\ell(x)^k$ where k is any real number (positive or negative). Here the no size-change condition requires

$$c(t) = \mathrm{Cov}(\ell, f)/\,\mathrm{Cov}(\ell, \ell^k). \tag{15}$$

Let us consider the case $g(\ell(x), t) = c(t)(\ell(x) - \mu(t))$ as another example. Here

$$\begin{aligned} \mathrm{Cov}(\ell, g) &= c(t)\,\mathrm{Cov}(\ell, \ell - \mu(t)) \\ &= c(t)\,\mathrm{Cov}(\ell, \ell) - c(t)\,\mathrm{Cov}(\ell, \mu(t)). \end{aligned}$$

But, $\mathrm{Cov}(\ell, \mu(t)) = 0$ and $\mathrm{Cov}(\ell, \ell) = \mathrm{Var}(\ell)$, so we end up with Equation (14) again (although the resulting penalty coefficient is then used in a different g).

What if we wanted $\mu(t)$ to follow, in expectation, a particular function $\gamma(t)$, e.g., the ramp $\gamma(t) = \mu(0) + b \times t$ or a sinusoidal function? The theory helps us also in this case. Adding $\mu(t)$ to both sides of Equation (10) we obtain:

$$\frac{\mathrm{Cov}(\ell, f) - \mathrm{Cov}(\ell, g)}{\bar{f} - \bar{g}} + \mu(t) \;=\; E[\mu(t+1)] \;=\; \gamma(t+1). \tag{16}$$

If g is a family of functions with a single degree of freedom (as is true of all the functions g considered above), then we can use this constraint to solve for the free variable. For example, if we want to control bloat with parsimony terms of the form $g(\ell(x), t) = c(t)\ell(x)^k$ we can substitute this into Equation (16), obtaining

$$\frac{\mathrm{Cov}(\ell, f) - c(t)\,\mathrm{Cov}(\ell, \ell^k)}{\bar{f} - c(t)E[\ell^k]} + \mu(t) \;=\; \gamma(t+1). \tag{17}$$

Solving for $c(t)$ gives:

$$c(t) = \frac{\mathrm{Cov}(\ell, f) - (\gamma(t+1) - \mu(t))\bar{f}}{\mathrm{Cov}(\ell, \ell^k) - (\gamma(t+1) - \mu(t))E[\ell^k]} \tag{18}$$

If $k = 1$, i.e., $g = c(t)\ell(x)$ (as in the standard parsimony pressure), this simplifies to

$$c(t) = \frac{\mathrm{Cov}(\ell, f) - (\gamma(t+1) - \mu(t))\bar{f}}{\mathrm{Var}(\ell) - (\gamma(t+1) - \mu(t))\mu(t)} \tag{19}$$

Note that, in the absence of sampling noise (i.e., for an infinite population), imposing that at each generation $E[\Delta\mu] = 0$ causes Equation (13) to reduce to $\mu(t) = \mu(0)$ for all

$t > 0$. However, in any finite population the parsimony pressure method can only achieve $\Delta\mu = 0$ *in expectation*, so there can be some random drift in $\mu(t)$ w.r.t. its starting value of $\mu(0)$. Experimentally we have found that this tends to be significant only for very small populations and long runs. If tighter control over the mean program size is desired, one can use Equation (18) with the choice $\gamma(t) = \mu(0)$, which leads to the following formula

$$c(t) = \frac{\mathrm{Cov}(\ell, f) - (\mu(0) - \mu(t))\bar{f}}{\mathrm{Cov}(\ell, \ell^k) - (\mu(0) - \mu(t))E[\ell^k]}. \tag{20}$$

Note the similarities and differences between this and Equation (15). In the presence of any drift moving $\mu(t)$ away from $\mu(0)$, this equation will actively strengthen the size control pressure to push back the mean program size to its initial value.[3]

As we will see in the following section, our technique gives users almost complete control over the dynamics of the mean program size, and control can be obtained in a single generation. It is thus possible to design interesting schemes where the covariance-based bloat control is switched on or off at different times, perhaps depending on the particular conditions of a run. In the next section we will, for example, test the idea of letting the GP system run undisturbed until the mean program size reaches a threshold, at which point we start applying bloat control to prevent further growth.

Finally, we note that while much of this theory assumes the use of fitness proportionate selection, Equation (6) is valid in general and one could imagine selection schemes that directly penalise the selection probabilities $p(\ell, t)$ rather than fitnesses. As we will see in the experiments, however, the penalty coefficients estimated using the theory developed for fitness proportionate selection actually work very well without any modification in systems based on other forms of selection, such as tournament selection.

5 Experimental Results

To verify the theory in a variety of different conditions, we conducted experiments using three different GP systems—two linear register-based GP systems and one tree-based GP system—and several problems. We briefly describe these systems and the problems in the next section, and then present some of our experimental results. Due to space limitations we will be able to report in detail on only a fraction of the tests we made.

5.1 GP Systems, Problems and Primitives

The first GP system we used is a linear generational GP system. It initialises the population by repeatedly creating random individuals with lengths uniformly distributed between 1 and 200 primitives. The primitives are drawn randomly and uniformly from a problem's primitive set. The system uses fitness proportionate selection and crossover applied with a

[3]We talk about size control pressure rather than parsimony pressure because $\mu(t)$ can drift both above and below $\mu(0)$.

rate of 100%. Crossover creates offspring by selecting two random crossover points, one in each parent, and taking the first part of the first parent and the second part of the second w.r.t. their crossover points. We used populations of size 100, 1,000 and 10,000. In each condition we performed 100 independent runs, each lasting 500 generations.

With this linear GP system we used two artificial test problems. The first was the `Hole` problem, which simply allocates a fitness of 0.001 to programs of size smaller than 10 nodes, and a fitness of 1.0 to all other programs. This problem was used because it presents the minimal conditions for bloat to occur (according to the crossover-bias theory described in Section 2). The second problem, which we will call `Square Root`, was one where the fitness of programs was simply the square root of their size, i.e., $f(x) = \sqrt{\ell(x)}$. This problem also satisfies the conditions for bloat, but, unlike the previous one, here the entire fitness landscape is expected to favour bloat (not just sections containing the very short programs) because the correlation between length and fitness is very high for all sizes. Because of its very strong tendency to bloat, we consider this problem a good stress-test our method.

For both `Hole` and `Square Root`, the fitness is determined completely by the size of the program, and arity is not an issue in a linear GP system, so any choice of primitive set produces the same results.

The second GP system we used is also linear and generational. It uses the same crossover (with the same rate) and the same form of initialisation as the first system, but initial program lengths are in the range 1 to 50. Runs lasted 100 generations. The system uses *tournament selection* (with tournament size 2) instead of fitness proportional selection. This allows us to test the generality of our method for controlling program size.

With this system we solved two classical symbolic regression problems. The objective was to evolve a function which fits a polynomial of the form $x + x^2 + \cdots + x^d$, where d is the degree of the polynomial, for x in the range $[-1, 1]$. In particular we considered degrees $d = 6$ and $d = 8$ and we sampled the polynomials at the 21 equally spaced points $x \in \{-1, -0.9, \ldots, 0.9, 1.0\}$. We call the resulting symbolic regression problems `Poly-6` and `Poly-8`. Polynomials of this type have been widely used as benchmark problems in the GP literature.

Fitness (to be maximised) was $1/(1 + \texttt{error})$ where `error` is the sum of the absolute differences between the target polynomial and the output produced by the program under evaluation over these 21 fitness cases. The primitive set used to solve these problems is shown in the first column of Table 1. The instructions refer to three registers: the input register `RIN` which is loaded with the value of x before a fitness case is evaluated and the two registers `R1` and `R2` which can be used for numerical calculations. `R1` and `R2` are initialised to x and 0, respectively. The output of the program is read from `R1` at the end of its execution.

The third GP system was a classical generational tree-based GP system using binary tournament selection, with subtree crossover applied with 100% probability. 35 independent runs were done for each of the five different targets for the average program size. The population size in each case was 2000, and each run went for 500 generations. The populations were initialized using the PTC2 tree creation algorithm [18] with the initial

Table 1: Primitive sets used in our experiments.

Polynomial	6-MUX
R1 = RIN	AND
R2 = RIN	OR
R1 = R1 + R2	NAND
R2 = R1 + R2	NOR
R1 = R1 * R2	
R2 = R1 * R2	
Swap R1 R2	

trees having size 150.

With the tree-based GP system we used the `6-Multiplexer` problem. This is a classical Boolean function indiction problem where the objective is to evolve a Boolean function with 6 inputs designed as `A0`, `A1`, `D0`, `D1`, `D2`, `D3` which produces as output a copy of one of the inputs `D0`–`D3`. These are know as the data lines of the multiplexer. The particular input copied over is determined by the inputs `A0` and `A1` (known as the address lines of the multiplexer), as follows: if `A0 = 0` and `A1 = 0` then `Out` = `D0`, if `A0 = 1` and `A1 = 0` then `Out` = `D1`, if `A0 = 0` and `A1 = 1` then `Out` = `D2`, if `A0 = 1` and `A1 = 1` then `Out` = `D3`. The function has 64 possible combinations of inputs, so we have 64 fitness cases. Fitness is the number of fitness cases a program correctly predicts. The primitive set used is shown in the second column of Table 1.

5.2 Results

We start by looking at the `Hole` and `Square Root` problems. As Figure 1(a)–(b) shows for populations of size 1,000, bloat is present in both cases, with the $\sqrt{\ell}$ fitness function bloating fiercely. Results for populations of size 100 and 10,000 are qualitatively similar.

To give a sense of the degree of control that can be achieved with our technique, Figure 2 illustrates the behaviour of mean program size for the `Hole` and `Square Root` problems when five different flavours of our size control scheme are used. Results are for populations of size 1,000, but other population sizes provide similar behaviours. Note the small amount of drift present when Equation (14) is used (first column). This is completely removed when instead we use Equation (20) (column 5, note the different scale, and also column 3 after transient). As columns 2 and 4 show, the user is free to impose any desired mean program size dynamics thanks to the use of Equation (19).

We turn to the `Poly-6` and `Poly-8` problems. As Figure 4(a)–(b) shows bloat is present in both problems. The behaviour of mean program size is brought under complete control with our technique as shown in Figure 3. Here, we used the same targets as in Figure 2 (although with slightly different parameters), but, to illustrate a further alternative, we used a parsimony term of the form $g(t) = c(t)/\ell$. This effectively promotes the shorter programs rather than penalising the longer ones.

Excellent size control was also obtained in the tree-based GP system when solving the

Poly Degree	Anti-bloat target	penalty	Success Rate
6	**none**		**0.77**
6	$\Delta\mu = 0$	$c\ell$	0.83
6	sin	$c\ell$	0.77
6	limit	$c\ell$	0.86
6	linear	$c\ell$	0.83
6	$\mu = \mu_0$	$c\ell$	0.83
6	$\Delta\mu = 0$	$c\ell^{-1}$	0.70
6	sin	$c\ell^{-1}$	0.77
6	limit	$c\ell^{-1}$	0.80
6	linear	$c\ell^{-1}$	0.79
6	$\mu = \mu_0$	$c\ell^{-1}$	0.71
8	**none**		**0.24**
8	$\Delta\mu = 0$	$c\ell$	0.37
8	sin	$c\ell$	0.47
8	limit	$c\ell$	0.41
8	linear	$c\ell$	0.36
8	$\mu = \mu_0$	$c\ell$	0.35
8	$\Delta\mu = 0$	$c\ell^{-1}$	0.26
8	sin	$c\ell^{-1}$	0.32
8	limit	$c\ell^{-1}$	0.26
8	linear	$c\ell^{-1}$	0.23
8	$\mu = \mu_0$	$c\ell^{-1}$	0.20

Table 2: Success rate comparison for `Poly-6` and `Poly-8` runs with different bloat control settings.

6-MUX problem, as illustrated in Figure 5. These use the same targets as in Figure 2, but again with slightly different parameters. Here the drift that's possible when using Equation (14) (the "Local" case in the figure) is quite apparent when compared to the very tight control obtained in the other configurations. When one considers, however, that there is no size limit or other form of bloat control being used, having the mean sizes in that case remain below 300 for 500 generations is still a significant achievement.

A performance comparison is not the focus of this paper. However, since virtually all bloat control methods need to strike a balance between parsimony and accuracy of solutions, it is reasonable to ask what sort of performance implications the use of our covariance-based bloat-control technique implies. As shown in Table 2 for the two polynomial regression problems, there is virtually no performance loss associated with the use of our technique.

6 Conclusions

For many years scientists, engineers and practitioners in the GP community have used the parsimony pressure method to control bloat in genetic programming. Although more recent and sophisticated techniques exist, parsimony pressure remains the most widely known and used technique.

The method suffers from two problems. Firstly, although good control of bloat can be obtained with a careful choice of the parsimony coefficient, such a choice is difficult and is often simply done by trial and error. Secondly, while it is clear that a constant parsimony coefficient can only achieve partial control over the dynamics of the average program size over time, no practical method to choose the parsimony coefficient dynamically and efficiently is available. The work presented in this paper changes all of this.

Starting from the size evolution equation developed in [24], we have developed a theory that tells us how to practically and optimally set the parsimony coefficient dynamically during a run so as to achieve complete control over the growth of the programs in a population. The method is extremely general, applying to a large class of control strategies of which the classical parsimony pressure method is an instance. Experimental results with three different GP systems, using different selection strategies and 5 different problems all strongly confirm the effectiveness of the method.

References

[1] L. Altenberg. The Schema Theorem and Price's Theorem. In L. D. Whitley and M. D. Vose, editors, *Foundations of Genetic Algorithms 3*, pages 23–49, Estes Park, Colorado, USA, 31 July–2 Aug. 1994. Morgan Kaufmann. Published 1995.

[2] P. J. Angeline. An investigation into the sensitivity of genetic programming to the frequency of leaf selection during subtree crossover. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 21–29, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[3] W. Banzhaf and W. B. Langdon. Some considerations on the reason for bloat. *Genetic Programming and Evolvable Machines*, 3(1):81–91, Mar. 2002.

[4] R. Crawford-Marks and L. Spector. Size control via size fair genetic operators in the PushGP genetic programming system. In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 733–739, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[5] S. Dignum and R. Poli. Generalisation of the limiting distribution of program sizes in tree-based genetic programming and analysis of its effects on bloat. In D. Thierens,

H.-G. Beyer, J. Bongard, J. Branke, J. A. Clark, D. Cliff, C. B. Congdon, K. Deb, B. Doerr, T. Kovacs, S. Kumar, J. F. Miller, J. Moore, F. Neumann, M. Pelikan, R. Poli, K. Sastry, K. O. Stanley, T. Stutzle, R. A. Watson, and I. Wegener, editors, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1588–1595, London, 7-11 July 2007. ACM Press.

[6] A. Ekart and S. Z. Nemeth. Selection based on the pareto nondomination criterion for controlling code growth in genetic programming. *Genetic Programming and Evolvable Machines*, 2(1):61–73, Mar. 2001.

[7] H. Iba. Complexity-based fitness evaluation for variable length representation. Position paper at the Workshop on Evolutionary Computation with Variable Size Representation at ICGA-97, 20 July 1997.

[8] H. Iba, H. de Garis, and T. Sato. Genetic programming using a minimum description length principle. In K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 12, pages 265–284. MIT Press, 1994.

[9] H. Iba, H. de Garis, and T. Sato. Temporal data processing using genetic programming. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 279–286, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[10] K. E. Kinnear, Jr. Evolving a sort: Lessons in genetic programming. In *Proceedings of the 1993 International Conference on Neural Networks*, volume 2, pages 881–888, San Francisco, USA, 28 Mar.-1 Apr. 1993. IEEE Press.

[11] K. E. Kinnear, Jr. Fitness landscapes and difficulty in genetic programming. In *Proceedings of the 1994 IEEE World Conference on Computational Intelligence*, volume 1, pages 142–147, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

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

[13] W. B. Langdon. The evolution of size in variable length representations. In *1998 IEEE International Conference on Evolutionary Computation*, pages 633–638, Anchorage, Alaska, USA, 5-9 May 1998. IEEE Press.

[14] W. B. Langdon. Size fair and homologous tree genetic programming crossovers. *Genetic Programming and Evolvable Machines*, 1(1/2):95–119, Apr. 2000.

[15] W. B. Langdon and R. Poli. Fitness causes bloat. In P. K. Chawdhry, R. Roy, and R. K. Pant, editors, *Soft Computing in Engineering Design and Manufacturing*, pages 13–22. Springer-Verlag London, 23-27 June 1997.

[16] W. B. Langdon and R. Poli. *Foundations of Genetic Programming*. Springer-Verlag, 2002.

[17] W. B. Langdon, T. Soule, R. Poli, and J. A. Foster. The evolution of size and shape. In L. Spector, W. B. Langdon, U.-M. O'Reilly, and P. J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 8, pages 163–190. MIT Press, Cambridge, MA, USA, June 1999.

[18] S. Luke. Two fast tree-creation algorithms for genetic programming. *IEEE Transactions on Evolutionary Computation*, 4(3):274–283, Sept. 2000.

[19] N. F. McPhee and J. D. Miller. Accurate replication in genetic programming. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 303–309, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[20] N. F. McPhee and R. Poli. Using schema theory to explore interactions of multiple operators. In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 853–860, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[21] 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.

[22] R. Poli. A simple but theoretically-motivated method to control bloat in genetic programming. In C. Ryan, T. Soule, M. Keijzer, E. Tsang, R. Poli, and E. Costa, editors, *Genetic Programming, Proceedings of the 6th European Conference, EuroGP 2003*, LNCS, pages 211–223, Essex, UK, 14-16 Apr. 2003. Springer-Verlag.

[23] R. Poli, W. B. Langdon, and S. Dignum. On the limiting distribution of program sizes in tree-based genetic programming. In M. Ebner, M. O'Neill, A. Ekárt, L. Vanneschi, and A. I. Esparcia-Alcázar, editors, *Proceedings of the 10th European Conference on Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, pages 193–204, Valencia, Spain, 11 - 13 Apr. 2007. Springer.

[24] R. Poli and N. F. McPhee. General schema theory for genetic programming with subtree-swapping crossover: Part II. *Evolutionary Computation*, 11(2):169–206, June 2003.

[25] G. R. Price. Selection and covariance. *Nature*, 227, August 1:520–521, 1970.

[26] J. Rosca. Generality versus size in genetic programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 381–387, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[27] J. Rosca. A probabilistic model of size drift. In R. L. Riolo and B. Worzel, editors, *Genetic Programming Theory and Practice*, chapter 8, pages 119–136. Kluwer, 2003.

[28] J. P. Rosca. Analysis of complexity drift 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 286–294, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[29] J. P. Rosca and D. H. Ballard. Complexity drift in evolutionary computation with tree representations. Technical Report NRL5, University of Rochester, Computer Science Department, Rochester, NY, USA, Dec. 1996.

[30] J. P. Rosca and D. H. Ballard. Rooted-tree schemata in genetic programming. In L. Spector, W. B. Langdon, U.-M. O'Reilly, and P. J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 11, pages 243–271. MIT Press, Cambridge, MA, USA, June 1999.

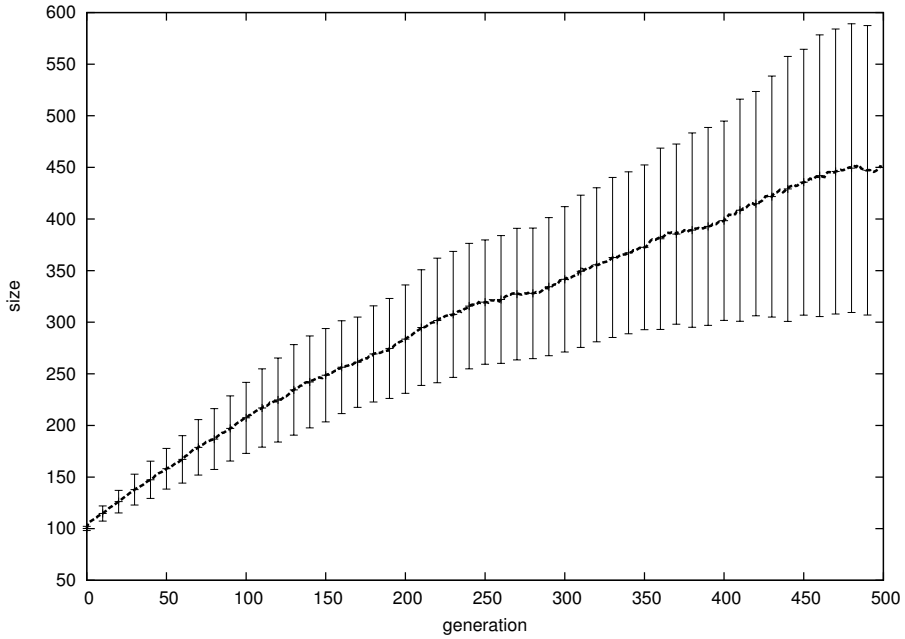
[31] T. Soule and J. A. Foster. Effects of code growth and parsimony pressure on populations in genetic programming. *Evolutionary Computation*, 6(4):293–309, Winter 1998.

[32] T. Soule and J. A. Foster. 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, 5-9 May 1998. IEEE Press.

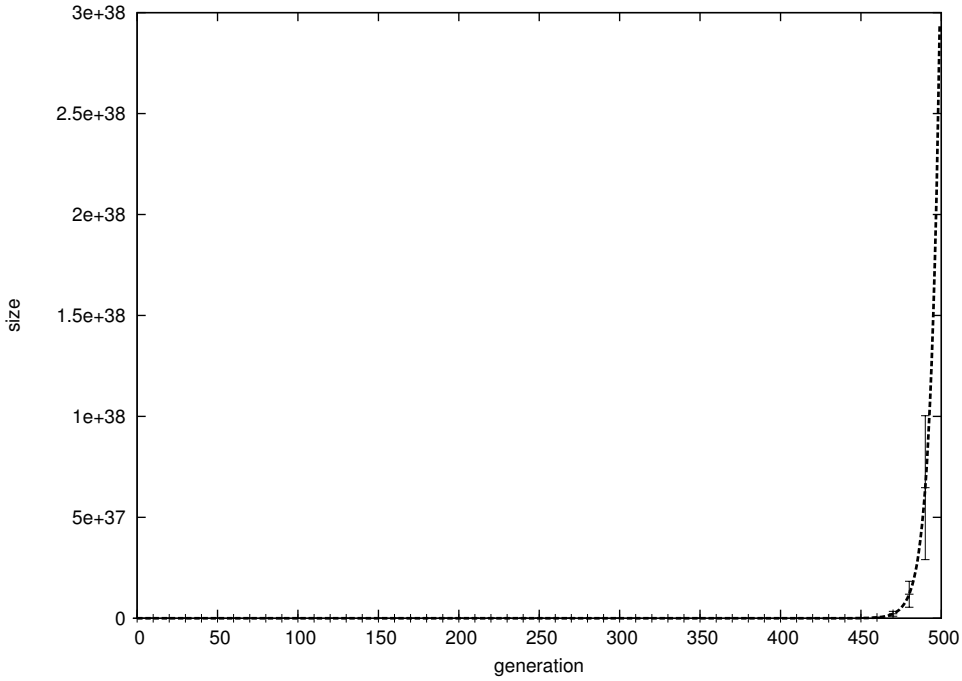
[33] B.-T. Zhang and H. Mühlenbein. Evolving optimal neural networks using genetic algorithms with Occam's razor. *Complex Systems*, 7:199–220, 1993.

[34] B.-T. Zhang and H. Mühlenbein. Balancing accuracy and parsimony in genetic programming. *Evolutionary Computation*, 3(1):17–38, 1995.

[35] B.-T. Zhang, P. Ohm, and H. Mühlenbein. Evolutionary induction of sparse neural trees. *Evolutionary Computation*, 5(2):213–236, 1997.



(a)



(b)

Figure 1: Behaviour of the mean program size in a linear GP system when solving the `Hole` problem (a) and the `Square Root` problem in the absence of bloat control for populations of size 1000. Results are averages over 100 independent runs. The error bars indicate the standard deviation across the runs. Note the log scale on plot (b).

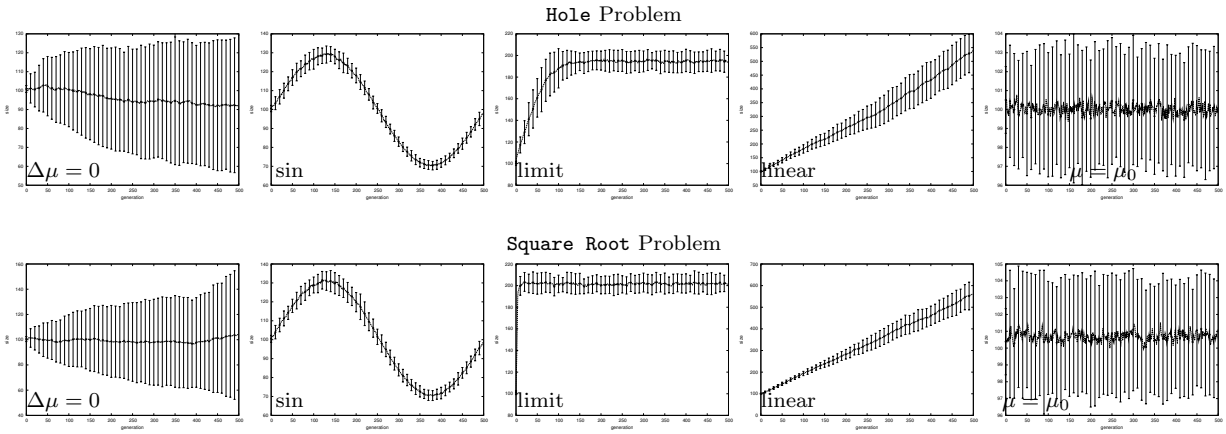


Figure 2: Size control obtained in the `Hole` and `Square Root` problems with populations of size 1,000 using a penalty function of the form $g(\ell(x), t) = c(t)(\ell(x) - \mu(t))$. The parameter $c(t)$ is computed according to Equation (14) so as to guarantee $E[\Delta\mu] = 0$ for the plots in the first column. Equation (19) was used for the remaining plots so as to guarantee $E[\mu(t)] = \gamma(t)$. $\gamma(t) = 30\sin(t/80) + \mu(0)$ in the second column, $\gamma(t) = t + \mu(0)$ in the fourth column and $\gamma(t) = \mu(0)$ in the fifth column. In the plots in the third column bloat control with $\gamma(t) = 200$ was activated only when mean program size reached 200. Results are the average mean size over 100 independent runs. Note that the range of the y-axes vary considerably across the plots.

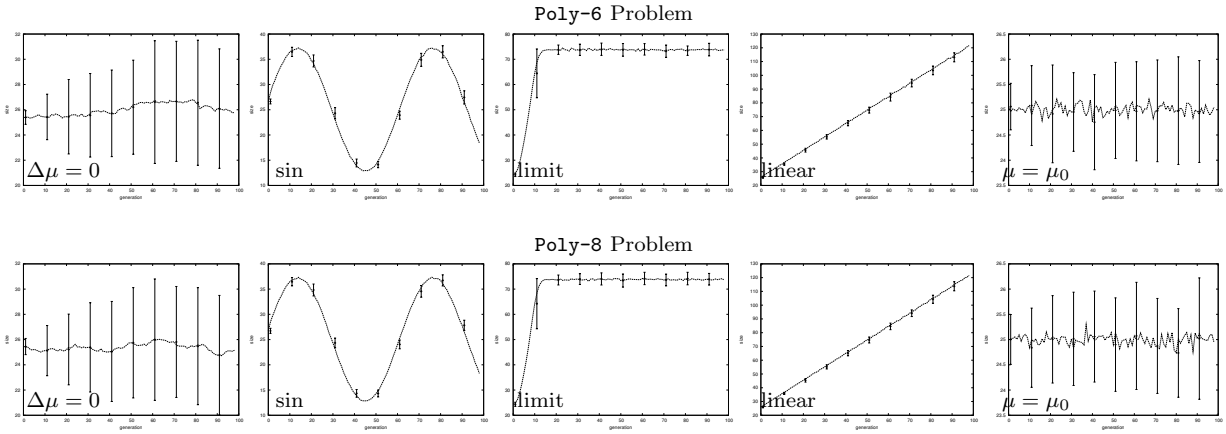
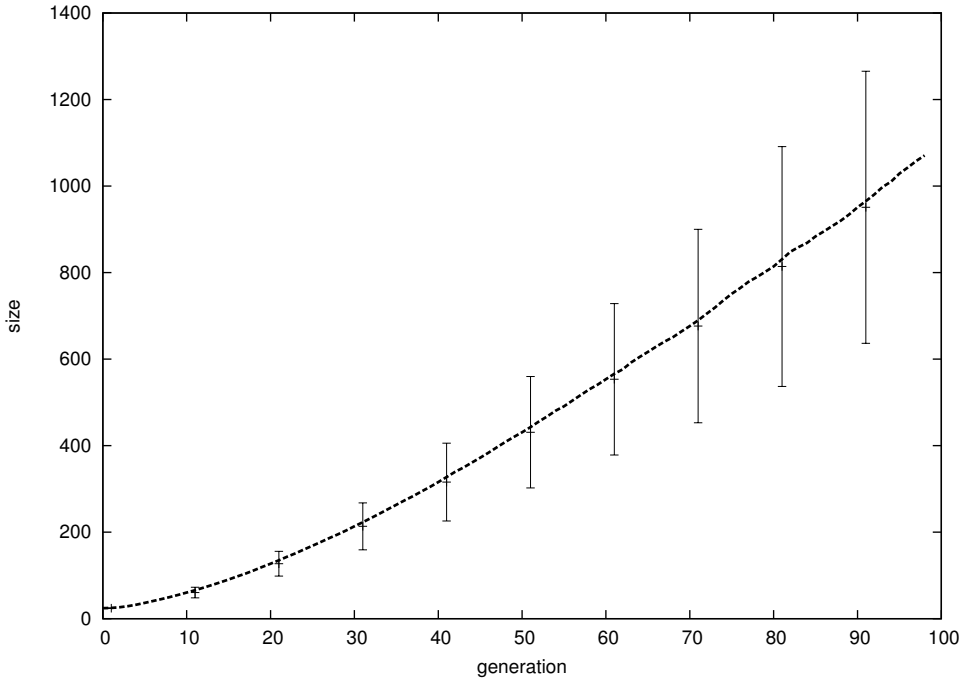
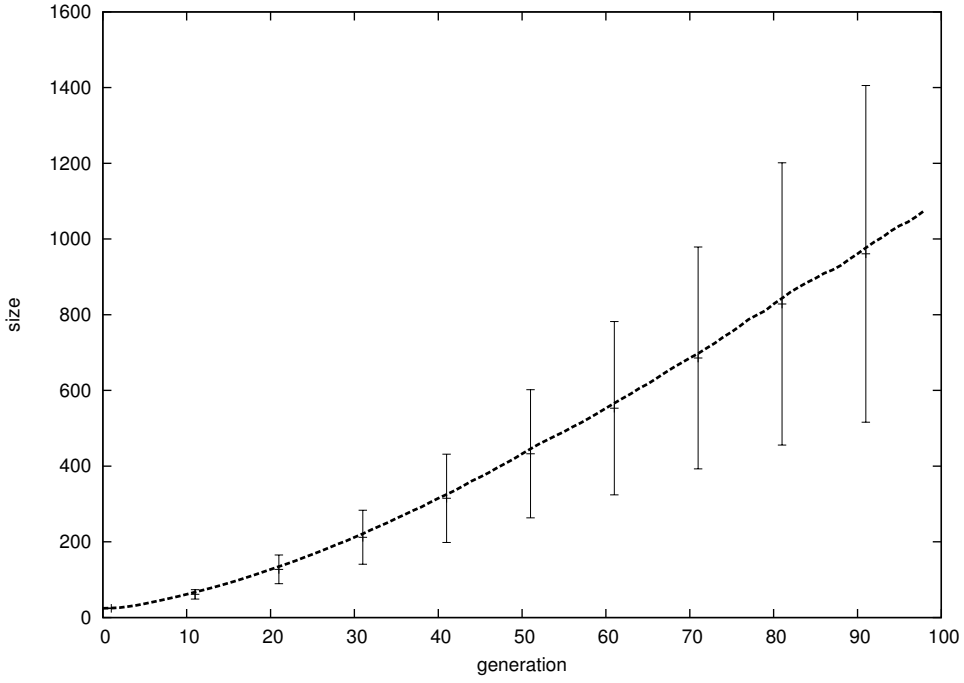


Figure 3: Size control obtained in the `Poly-6` and `Poly-8` problems with populations of size 1,000 using a penalty function of the form $g(\ell(x), t) = \frac{c(t)}{\ell(x)}$. The parameter $c(t)$ is computed according to Equation (15) with $k = -1$ so as to guarantee $E[\Delta\mu] = 0$ for the plots in the first column. It was computed according to Equation (18) (with the same k) so as to guarantee $E[\mu(t)] = \gamma(t)$ for the remaining plots. $\gamma(t) = 12.5\sin(t/10) + \mu(0)$ in the second column, $\gamma(t) = t + \mu(0)$ in the fourth column and $\gamma(t) = \mu(0)$ in the fifth column. In the plots in the third column bloat control with $\gamma(t) = 75$ was activated only when mean program size reached 50. Results are the average mean size over 100 independent runs. Population sizes 100 and 10,000 provided qualitatively similar behaviours. Note that the range of the y-axes vary considerably across the plots.



(a)



(b)

Figure 4: Behaviour of the mean program size in a linear GP system when solving the `Poly-6` problem (a) and the `Poly-8` problem in the absence of bloat control for populations of size 1,000. Results are averages over 100 independent runs.

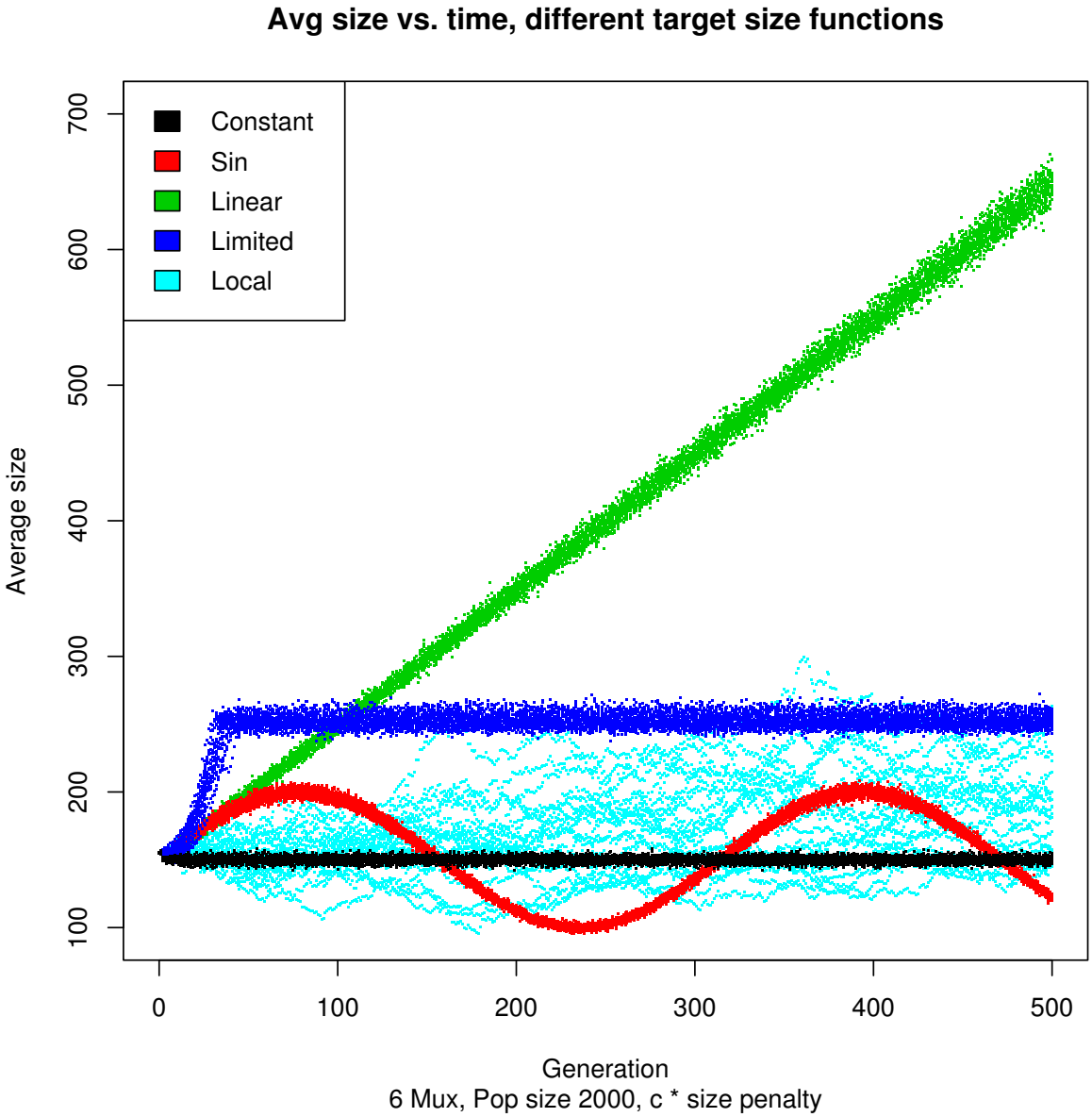


Figure 5: Scatterplot of the average size over multiple runs of the 6-MUX problem with various size target functions. In all cases the population size was 2000 and we used the penalty function $f - c \times \text{size}$. The "Constant" case had a constant target size of 150. The "Sin" had a target size function of $\sin((\text{generation} + 1)/50.0) \times 50.0 + 150$. The "Linear" case had a target function of $150 + \text{generation}$. The "Limited" case used no size control until the size exceeded a set limit (250 in this case), after which a constant target of 250 was used. The "Local" case used a target of $\Delta\mu = 0$, i.e., the average size in each generation should be approximately equal to the average size of the previous generation.