Effects of Code Growth and Parsimony Pressure on Populations in Genetic Programming *

Terence Soule
Computer Science Dept.
St Cloud State University
139 Engineering and Computing Center
St. Cloud, MN 56301-4498
tsoule@eeyore.stcloudstate.edu
(320) 255-4966

James A. Foster
Laboratory for Applied Logic
Computer Science Dept.
University of Idaho
Moscow, Idaho 83844-1010
foster@cs.uidaho.edu

Abstract

Parsimony pressure, the explicit penalization of larger programs, has been increasingly used as a means of controlling code growth in genetic programming. However, in many cases parsimony pressure degrades the performance of the genetic program. In this paper we show that poor average results with parsimony pressure are a result of "failed" populations which overshadow the results of populations which incorporate parsimony pressure successfully. Additionally, we show that the effect of parsimony pressure can be measured by calculating the relationship between program size and performance within the population. This measure can be used as a partial indicator of success or failure for individual populations.

Keywords: Code Growth, Code Bloat, Parsimony, Genetic Programming, Introns

*This work was supported by funding from the URO seed grant program.

1 Introduction

The use of parsimony pressure as a means of controlling the size of programs generated with genetic programming (GP) has grown considerably in recent years. In many cases parsimony pressure appears to have been added as means of controlling code growth without careful consideration of the possible side effects of its use or the fact that it has been known to degrade performance in some applications. In large part, this oversight is because the effects of parsimony pressure on an evolving population, beyond limiting code growth, are not well understood, nor are the reasons for its failures.

The tendency of programs generated with GP to grow extremely large without corresponding increases in fitness is well documented in the GP literature [?, ?, ?, ?, ?, ?, ?]. Most code growth consists of code which does not directly contribute to a program's performance.

Code growth is less well documented in areas of evolutionary computation other than GP. However, code growth has been demonstrated in a few other evolutionary paradigms [?, ?, ?]. Further, the research on code growth in GP strongly suggests that it will be a problem in any evolutionary technique which uses variable size representations. Most variable size representations are vulnerable to the same causes of code growth that are believed to affect GP.

Code growth is a significant problem, as rapid program growth consumes considerable resources without directly contributing to a solution. Additionally, non-functional code may interfere with finding better solutions, since most of the code manipulation by evolutionary operators will occur in the non-functional regions.

In addition to being a serious problem, code growth is an interesting phenomenon which demonstrates the strengths of evolutionary techniques. Code growth is believed to be a protective response to the destructive effects of crossover and mutation. Thus, the evolutionary process is capable of recognizing and evolving solutions to problems which the designer did not anticipate. Clearly this is a highly desirable feature for a machine learning system.

Direct penalization of larger programs (parsimony pressure) is an obvious mechanism for limiting code growth. Parsimony pressure should take advantage of the evolutionary process itself to produce programs which are both successful and succinct. However, experiments using parsimony pressure have had mixed results. In some instances the desired parsimonious solutions were obtained [?, ?, ?], but in other instances the performance of the programs was clearly degraded by the parsimony requirement [?, ?, ?]. This conflict has far reaching consequences. If GP is not generally capable of balancing two distinct, and possibly conflicting, requirements then it has limited usefulness as a general problem solving technique.

This paper shows that parsimony pressure can produce poorer performance. The amount of degradation being dependent upon the distribution of individuals within a population. Although the results do suggest general guidelines for designing robust forms of parsimony pressure our primary goal is to understand how balancing the twin goals of performance and parsimony affects the evolutionary process.

2 Code Growth

2.1 Terminology

To discuss code growth in detail some terminology regarding the types of code found in GP generated programs will be helpful. In hand written code each instruction is expected, under some set of circumstances, to be executed and to serve some useful purpose. However, this is not the case for GP generated code. GP code is normally saturated with instructions which are never executed or which, when executed, serve no useful purpose and do not contribute to the individual's fitness.

The GP literature commonly divides sections of code into introns and exons.

Informally introns are sections of code which do not contribute to fitness and exons are sections of code which do contribute. However, these terms are often used without precise definitions. The definitions which have been presented in the literature are not always compatible. Furthermore, the terms intron and exon are borrowed from the biological community where they have somewhat different meanings than when applied to the evolution of programs. We have found it useful to make a further distinction between regions of code which can *possibly* contribute to a program's output, but happen not to, as opposed to regions which can not contribute under any circumstances. To avoid adding to the confusion surrounding these terms we chose to introduce two new terms defined as follows:

Definition 1 *A node n in a program's syntax tree is* operative *if the removal of the subtree rooted at that node will change the program's output on some input. Conversely a node is* inoperative *if it is not operative.*

Definition 2 *A node n in a program's syntax tree is* viable *if there exists a tree such that replacing the tree rooted at node n with the new subtree will change the program's output on some input. Conversely a node is* inviable *if it is not viable.*

Notice that with these definitions inviable code is a proper subset of inoperative code. Thus, any program will have at least as much inoperative code as it does inviable code.

Expanding these definitions to include non-tree genome shapes is not difficult.

2.2 Causes of Code Growth

Several independent theories have been proposed to explain code growth. In roughly equivalent theories, Nordin and Banzhaf [?], McPhee and Miller [?], and Blicke and Thiele [?] argue that code growth occurs to protect programs against the potentially destructive effects of evolutionary operations like crossover and subtree mutation. Intuitively, removing a section of code from a program and replacing it with randomly chosen code from another source would rarely be expected to increase the program's performance. Several studies have confirmed this intuition by showing that few crossover events increase fitness and many lower fitness [?, ?]. Thus, on average crossover is a destructive or neutral operation.

The number of possible destructive operations decreases as more of the code within a program becomes inviable. In this case, code modifying operations are more likely to change the inviable code and, by definition, output and fitness will not be affected. Thus, an evolutionary advantage for growing inviable code should exist.

To a more limited extent this argument may apply to inoperative code. Although inoperative code does not contribute to fitness, changing it may change the program's output. Thus, it is less clear that increased inoperative code will protect programs against change in the same manner as inviable code.

A second feature of these theories is that operative code should be minimized. Smaller sections of operative code are less likely to be affected, and thereby damaged, by crossover, providing another evolutionary benefit.

Several studies with non-destructive (hill-climbing) versions of crossover have shown that they lead to much lower rates of overall code growth [?, ?, ?, ?]. This is strong evidence that code growth occurs, at least in part, to protect against crossover's destructive effects. In addition, non-destructive crossover does not appear to decrease the amount of operative code [?]. Thus, the ratio of operative to non-operative code is much lower with destructive corssover, further evidence that code growth is a mechanism for protecting operative code.

A second potential cause of code growth is *removal bias* [?, ?]. Again the general destructiveness of code modifying operations is the root cause of this form of code growth. In general, removing a smaller branch from a program's syntax tree as a part of crossover, subtree mutation, or a similar operation, makes it more likely that only inviable code will be affected and thus that the operation will be fitness neutral. However, if the operation does occur in inviable code then the replacement branch will not affect performance, regardless of its size. Thus, there is a bias towards programs which have a small branch removed, but not a corresponding bias in favor of programs that had a small branch added. The net effect is a general growth in the inviable sections of a program's code. Although removal bias is most clearly applicable to inviable code it may also apply, probably to a somewhat lesser extent, to inoperative code.

Langdon and Poli have argued that code growth is partially caused by the distribution of semantically equivalent solutions in the solution space [?]. In most variable length representations a particular solution can be represented by many semantically equivalent, but syntactically different, programs. The existence of inviable and inoperative code guarantees that for any given program size there are many more larger versions of the solution than there are smaller ones. Langdon and Poli argue that as a search progresses it is more likely to find progressively larger solutions because there are more of them. Thus, it is possible that some code growth is simply due to the distribution of solutions in the search space, rather than to an evolutionary influence.

These are three distinct causes of code growth, each of which is capable of causing code growth by itself. However, it is likely that under normal circumstances they work in concert. For example, removal bias may produce inviable code which is then conserved for protective purposes.

In addition to these relatively neutral roles for inviable code, it is also possible that inviable code acts more directly in producing better solutions. Inviable code may act as a storage area for building blocks which are later moved back into the viable regions of code. Alternatively, inviable code may serve as "scratch paper" in which building blocks are created before being transfered into viable code. In either case, additional inviable code could improve the evolutionary process, creating an evolutionary advantage for programs with more inviable code and leading to code growth.

Both of these possibilities rely on the mechanism of crossover to transfer code back and forth between viable and inviable regions or on mutation to turn introns on and off (for example, by changing an `if(false)` into `if(true)`). Thus, recent experiments showing that crossover is not necessarily important in GP [?, ?, ?, ?] cast serious doubt on these theories. We are also suspicious of storage and "scratch paper" hypotheses because they require a kind of evolutionary foresight to determine which partial solutions to store, or which scratch manipulations to make. Rather it seems that immediate selective pressure is the major force directing evolution.

2.3 Solutions to Code Growth

A large number of methods for controlling code growth have been proposed. Although this paper will focus on parsimony pressure it is helpful to consider some of the other methods.

Probably the most common approach to controlling code growth is to set a fixed limit on program size, either by limiting the number of instructions in the program or, in the case of syntax trees, by limiting the allowed depth of those trees. Programs exceeding the limits are removed from the population. Because the size of a program is easily calculated during evaluation this approach requires relatively little additional computation. However, recent studies have shown that depth limits can interfere with the creation of good solutions [?, ?]. In addition, an appropriate size limit must be chosen in advance. A size limit which is too small may make it difficult or impossible for a solution to evolve, whereas an overly generous limit will still allow extraneous growth. Clearly it would be preferable to allow the GP to evolve the appropriate size. This is the goal of the parsimony pressure approach.

Recently, non-destructive or hill-climbing versions of crossover have been shown to reduce the rate of code growth in GP [?, ?, ?]. In non-destructive crossover the offspring of a crossover event is preserved only if its fitness is greater (or possibly equal) to its parent's fitness. It is hypothesized that these methods work by reducing or eliminating the evolutionary advantage of protective code. This means that causes of code growth not directly related to protective code, such as those suggested by Langdon and Poli, will not be affected. Additionally, these forms of crossover limit GP to a hill-climbing search which may be insufficient for solving more difficult problems.

Nordin, Francone, and Banzhaf have suggested the using *explicitly defined introns* (EDIs) to reduce the impact of code growth and to improve GP search [?]. An explicitly defined intron is an instruction which has no semantic effect, but does influence the probability of crossover. In tree based GP, EDIs can be represented by nodes which have a variable chance of being selected for crossover, but no other effect. EDIs can protect against destructive operations in the same manner as normal inviable code. A few EDIs with high crossover probabilities make it much less likely that viable code will be affected by crossover, thus serving the same function as large sections of inviable code. In addition, because they are explicitly defined, EDIs are easily recognized and removed from the final program. Thus, EDIs have several distinct advantages over normal GP generated inviable code. EDIs have been shown to be particularly effective at controlling code growth when combined with parsimony pressure [?, ?].

There are several reasons for focusing on parsimony pressure as a means of controlling code growth. First, several of the other methods are special cases of parsimony pressure or incorporate some parsimony pressure. For example, fixed limits are a form of parsimony pressure where the parsimony function ($p(s_i)$) is a step function. EDIs are commonly used with parsimony pressure as an additional limiting factor. Thus, an understanding of parsimony pressure should improve our understanding of these techniques as well. Second, parsimony pressure is a very general approach and should control growth regardless of its cause. Third, parsimony pressure is easily applied, requiring very little overhead. Fourth, and perhaps most importantly, parsimony pressure seems to be the natural method for controlling code growth in GP. It uses evolution itself to produce solutions which are both successful and succinct. If GP is found to be generally incapable of balancing the twin requirements of performance and parsimony this would indicate a serious limitation of what it can accomplish.

Parsimony pressure uses a fitness penalty based on size to deter code growth. Formally, the fitness function with parsimony pressure for a program i is:

$$f(i) = P(i) - p(s_i) \tag{1}$$

where $P(i)$ measures how well individual i actually performs, s_i is the size of i, and p is the function defining the amount of parsimony pressure. Often p is a simple linear function and the fitness function is:

$$f(i) = P(i) - \alpha s_i.$$

Parsimony pressure has been quite effective in some cases [?, ?, ?] and a failure in others [?, ?], but the cause of these different results is unclear.

Zhang and Muhlenbein have proposed an adaptive parsimony approach in which the parsimony factor α is a function of the performance and size of the best individual from preceding generations [?]. Blickle used adaptive parsimony successfully on two regression problems [?]. One limitation of this method is that it requires an error limit which must be predefined by the user. A poor choice for this parameter can hinder performance. A further drawback of this approach over fixed parsimony pressure is that it requires additional overhead to calculate the new parsimony value at each generation.

3 Sample Applications of Linear Parsimony Pressure

We chose to use the even parity problem (with 6 inputs) as our test problem because it is commonly used in GP (see, for example [?, ?, ?, ?]) which makes it easier to compare our results to those of other researchers. Additionally, previous work with the even parity problem has shown that a simple GP, such as we are using, is unlikely to be successful on this problem with 6 inputs [?, ?]. Thus, we can be reasonably sure that for each trial the GP can always make further improvements and is not limited by having found the perfect solution.

The details of this problem and our parameters are shown in Table 1.

We performed separate trials using a linear parsimony function with the parsimony coefficients (α) of $0.0, 0.01, 0.02, 0.05$, and 0.1. Linear parsimony is used because it is the simplest form of parsimony, is commonly used, and is most amenable to analysis. However, the results can be generalized to other parsimony functions and, to a more limited extent, to adaptive forms of parsimony pressure.

The size of a program is the total number of nodes it contains. We ran one hundred trials for each test case and each trial ran for 75 generations, except in the case of no parsimony pressure where the rapid code growth forced a halt at 50 generations. Note that for the lowest level of parsimony pressure ($\alpha = 0.01$) the fitness improvement for correctly classifying one additional input case outweighs the parsimony penalty for 99 additional nodes. Thus, at this level of parsimony pressure the main effect will be to distinguish between programs of equal performance.

The average program size (over all 100 trials) for the five levels of parsimony pressure are shown in Figure 3. This figure clearly shows that increasing amounts of parsimony pressure decrease the amount of code growth. The differences in size become significant (two-tailed test, $P < 0.01$) after one generation between any two of the parsimony levels.

Table 1: Summary of the Even-N-Parity problem.

Objective	Determine whether the parity of 6 boolean inputs is even or odd
Terminal set	The 6 input values
Function set	`AND`, `NAND`, `OR`, and `XOR`
Fitness	Number of correctly categorized cases (out of 2^N)
Selection	stochastic remainder
Population size	500
Initial population	random trees
Parameters	66.6% crossover, no mutation, results averaged over 100 trials
Termination	50 or 75 generations
# of Trials	100

Figure 3 shows the average performance (averaged over all 100 trials) for the five levels of parsimony pressure. Here it is clear that increasing the amount of parsimony pressure degrades the average performance of the genetic program. The differences in performance take longer to become significant than the differences in size. However, by generation 19 all of the differences are significant (two-tailed test, $P < 0.01$). The two most extreme trials ($\alpha = 0.0$ versus $\alpha = 0.1$) become significantly different by generation 8 (two-tailed test, $P < 0.01$). The best performance (averaged over all 100 trials) were, 48.82 for no parsimony (generation 50), 50.83 for $\alpha = 0.01$ (generation 75), 48.76 for $\alpha = 0.02$ (generation 75), 41.95 for $\alpha = 0.05$ (generation 75), and 35.14 for $\alpha = 0.1$ (generation 75).

These results are similar to other negative results with parsimony pressure [?, ?]. Program size is controlled, but performance suffers. Both the benefits and costs of parsimony pressure are clearly correlated with the amount of parsimony pressure used.

The initial rise in performance between generations 0 and 7 is similar for all five cases and is believed to occur as selection moves the population average towards the best individual in the initial, random population. Thus, because this increase does not depend on the production of improved solutions, it does not truly reflect the full evolutionary process and probably shouldn't be interpreted as an early success for the more restrictive amounts of parsimony pressure.

Although in these trials the average effect of parsimony pressure is clearly to decrease performance, as well as code size, the process of averaging over multiple trials obscures contrary and interesting behaviors in the individual populations. In many cases the parsimony pressure drives the entire population to the minimal possible size (a single node for these experiments) without regard for the resulting performance. We will refer to these populations as *failed* populations to reflect that no improvement was made beyond the solutions in the initial population. Table 2 separates the trials based on the average performance at generation 75 (generation 50 for the trial without any parsimony pressure). The table includes the average performance, and standard deviations for all trials and for the subset of the trials in which performance exceeded 35. (For this problem random guessing scores 32, achievable by a program with a single node. Scores lower than 32 require a larger program. Thus, it is reasonable to treat 32 as the minimal performance and any program which cannot achieve performance of at least 35 can be

Table 2: Distribution of trial results at generation 75 (50 for $\alpha = 0.0$) for different levels of parsimony pressure.

Ave. Performance Range	# of Trials $\alpha = 0.0$	# of Trials $\alpha = 0.01$	# of Trials $\alpha = 0.02$	# of Trials $\alpha = 0.05$	# of Trials $\alpha = 0.1$
≤ 35	0	1	6	46	86
$35 - 37$	0	3	7	6	3
$37 - 40$	7	6	17	11	1
$40 - 43$	19	19	17	9	4
$43 - 46$	25	26	15	10	1
$46 - 49$	23	12	10	3	3
$49 - 52$	12	9	16	13	1
$52 - 55$	6	11	5	2	1
$55 - 58$	3	9	7	0	0
> 58	5	4	0	0	0
Ave. Performance	46.55	46.69	44.16	38.31	33.63
Standard Deviation	5.07	6.09	6.57	7.02	4.43
Ave. Performance Discounting < 35 case.	46.55	46.84	44.90	43.60	43.26
Standard Deviation Discounting < 35 case.	5.07	5.94	6.05	5.47	5.74

Table 3: Results of applying a chi-squared test to the non-failed trials.

P	χ^2
0.00	6.24
0.01	4.80
0.02	2.95
0.05	9.24
0.1	–

considered a failure.)

The separated results show that if the trials with a performance below 35 are omitted the remaining trials show a minimal decline in performance with parsimony pressure. Of the trials using parsimony pressure only the trial with $\alpha = 0.01$ has a performance which is significantly better than the other trials (two-tailed test, $P < 0.05$). Further, the distribution of the non-failed trials is similar, suggesting that evolution is occurring normally. We believe that with higher levels of parsimony pressure the trials generally produce a bimodal distribution with a sharp peak for the failed trials and a more normal distribution for the successful trials.

A chi-squared test was applied to the non-failed trials to determine if their distribution was in fact normal. The results of this test are shown in Table 3. For the $\alpha = 0.1$ case the number of non-failed tests was insufficient to give statistically meaningful results. Four bins were used in the test so χ^2 values on the order of four suggest a normal distribution.

These results strongly suggest that the non-failed trials are evolving normally. While the χ^2 values are not entirely indicative of a normal distribution they are very close to the values for the parsimony free case. Thus, it is reasonable to conclude that the fitnesses form a bimodal

Table 4: Average program sizes at generation 75 (50 for $\alpha = 0.0$) for different levels of parsimony pressure.

	$\alpha = 0.0$	$\alpha = 0.01$	$\alpha = 0.02$	$\alpha = 0.05$	$\alpha = 0.1$
Avg. Size	374.32	103.56	77.42	32.86	7.80
Standard Deviation	138.60	41.74	30.30	29.63	16.03
Avg. Size Only ≤ 35 case.	374.32	1.00	16.03	3.38	1.77
Standard Deviation Only ≤ 35 case.	138.60	–	23.83	7.17	2.75

distribution with one peak for the failed trials and a second, more normal, distribution for the successful trials.

This is very strong evidence that most of the decrease in performance seen in Figure 3 is caused by those trials in which the entire population is trapped at a minimal, or near minimal, fitness. It further appears that populations which escape this trap show comparatively little degradation. Thus, in some cases GP is able to balance performance and parsimony, but when it fails to achieve a balance the presumably simpler task of evolving minimal programs is favored.

Table 4 gives the average sizes and standard deviations for all trials and only the failed trials. Not surprisingly, the subset of unsuccessful trials shows much smaller average program sizes than for all of the trials. In general, this effect increases with increasing parsimony pressure. Thus, as with performance, it is clear that much of the size differences attributable to high levels of parsimony pressure are caused by the failed trials. However, unlike performance, the average size difference between trials with parsimony pressure and trials without parsimony pressure is quite large even when only the successful trials are considered.

Figure 3 shows a slow, but constant, increase in average performance even with the strongest levels of parsimony pressure we used. The absence of a decrease in performance implies that the failed populations are trapped almost immediately. If they did evolve successfully for a time before being pulled down to the minimal performance we would observe a decrease in performance.

Averaging over multiple trials is clearly obscuring the fact that some trials evolve nearly normally in the presence of parsimony pressure. Given these observations it is reasonable to ask why are some populations adversely affected by parsimony pressure while others are relatively unaffected?

4 An Analysis of Parsimony Pressure

With parsimony pressure the fitness of an individual is a function of both its performance and its size. Performance is the raw measure of how well a program performs, as opposed to the fitness which also incorporates the parsimony penalty. A simplified fitness landscape can be constructed using performance and program size as the two major axes. We believe that the distribution of the individuals of a population within this landscape can be used as an indicator of whether a particular population will successfully evolve in the presence of a parsimony function.

A major feature of the simplified landscape are the regions of equal fitness. These are a series of contours, each defining an area of performance and size in which the programs have an identical fitness. The equation defining a contour of constant fitness is derived by making fitness a constant in Equation 1:

$$P(i) = f_i + p(s_i).$$

These contours have the same shape as the parsimony function ($p()$). So, a linear parsimony function produces linear contours with slopes equal to the coefficient α of the parsimony function, as is illustrated in Figure 3. More complex parsimony functions would produce correspondingly more complex regions of equal fitness.

00.55somepars.eps

Figure 3: The simplified landscape created with linear parsimony pressure.

The difference in fitness between two individuals is proportional to the distance between them along an axis perpendicular to the contours of constant fitness. In contrast, two individuals on the same contour have the same fitness (although very likely different performances and sizes) and so there is no relative selection pressure between them. Thus, selection pressure is strongest perpendicular to the contours. Over time selection should move a population through the landscape in the direction perpendicular to the contours and towards the contours representing the highest fitness.

The affect of parsimony pressure on a population depends on the relationship between size and performance within the population. For simplicity we assume that this is a linear relationship (we will present further justification for this assumption later) so that the population can be roughly described by the equation

$$performance = a * size + b.$$

It is now possible to compare a linear parsimony function with the population distribution by comparing the values of α (the amount of parsimony pressure) and a (the average relationship between size and performance in the population).

Figure 4 illustrates the two interesting and expected cases: $a > \alpha > 0$ and $\alpha > a > 0$. (The case $\alpha < 0$ implies reverse parsimony pressure is being used to favor larger programs, which is unlikely. The case $a < 0$ means there is a negative correlation between size and performance in which case selection will naturally prefer the smaller programs and parsimony pressure would not be necessary.) Figure 4 shows that for $\alpha < a$ selection will be strongest for the better performing individuals, but for $\alpha > a$ selection will be strongest for the smaller individuals. Thus, the distribution of the population helps determine whether performance or size is the primary factor in determining survival.

Next we consider the expected relationship between size and performance within a population. Since an evolving population tends to converge, the fitness of the individuals within the population will move towards a region of equal fitness. This means that all of the individuals will tend to fall along one of the contour lines, resulting in a population which is roughly aligned with the contours and producing a positive correlation between size and performance. Because this correlation is aligned with the parsimony function, a linear parsimony function should result in a population with a roughly linear correlation between size and performance and the values of α and a should be equal.

Of course, this is only a first approximation of a populations distribution. Given the randomizing affects of crossover it would be unreasonable to expect a perfectly linear relationship between size and fitness. However, we will show that this approximation is sufficient to make predictions about how a population will respond to parsimony pressure. Additionally, if the parsimony function is not linear, or there are other reasons to expect a relationship other than a linear one, it is possible to calculate the appropriate coefficients for the expected functional relationship instead of the linear coefficient used here.

The tendency of populations to align with the contours of constant fitness defined by the parsimony function is the primary influence on a population's distributions. However, all three of the theories of code growth predict a secondary effect on the population distribution.

If code growth protects against the destructive affects of crossover (and similar evolutionary operators), then after crossover the largest programs are most likely to have retained their fitness. If removal bias occurs then the largest programs resulting from crossover are the least likely to have been damaged. If Langdon and Poli's hypothesis is correct then the largest programs resulting from crossover are mostly likely to be semantically equivalent to their predecessors. All three theories predict that the largest programs are the most likely to retain their fitness. Thus, all of three theories predict that, in general, an additional relationship between size and performance exists which leads to $a > \alpha$. It is important to realize that this does not mean that larger programs are better at solving the problem, rather they are more likely to retain a previously discovered solution.

Figure 4 shows the relationship between size and performance (the value a) at each generation averaged over 100 trials. In some trials the population completely converged (all programs were identical in size and performance) which led to an undefined correlation. These trials became more frequent with increased parsimony pressure and generally were the cases in which the population converged to the minimal size. We omitted individual trials from these averages when their correlation became undefined. This omission meant that by the final generation only 25 trials contributed to the average for $\alpha = 0.1$, 63 trials contributed for $\alpha = 0.05$, 96 contributed trials for $\alpha = 0.02$, 99 contributed trials for $\alpha = 0.1$, and all 100 trials contributed for $\alpha = 0.0$.

As one would expect Figure 4 shows that the correlation between size and performance is zero at generation zero, since a larger random program should not be better than a smaller random program. However, as evolution proceeds the correlation increases until $a > \alpha$ as predicted. This is an encouraging result for the effective use of parsimony pressure because, as previously noted, $a > \alpha$ implies that selection is strongest in the direction of improved performance and size becomes a secondary factor.

Figure 4 also helps explain why trials failed when parsimony pressure was applied. Although the ideal relationship of $a > \alpha$ eventually arises, for the first few generations the opposite case holds. Furthermore, there is a clear relationship between the number of generations during which $a < \alpha$ and the number of trials which fail. For $\alpha = 0.0$ and 0.01 the condition $a < \alpha$ is true only in the initial generation and there are almost no failed populations. For $\alpha > 0.01$ the condition $a < \alpha$ is true for several more generations and there are many more failed populations.

This also suggests that most forms of adaptive parsimony pressure should work well, as long as the amount of pressure is changed relatively slowly. A slowly changing parsimony function gives the population time to adjust its distribution to maintain the optimal $a > \alpha$ condition.

As expected, the relative difference between α and a (when $a < \alpha$) also appears to influence the probability that a populations will fail. As more parsimony pressure is applied a lags further behind α and more of the populations fail. This is because the larger the difference between α and a the more strongly selection favors small size over good performance.

Figure 4 shows the correlation between the value a for the populations during the critical early generations and the average performance of the populations in the final generation. In the initial, random population this correlation is quite low for all test cases. However, as the populations begin to evolve the correlations increase, particularly for the cases where stronger parsimony pressure is used. This makes it clear that when parsimony pressure is used, the relationship of size to performance in a population in the early generations has a significant effect on the performance of that population 70 generations later. To our knowledge this is the first measurement which can predict the eventual success or failure of a population based on its behavior in its earliest generations.

The size to performance relationship is clearly a key factor in determining whether a population subjected to parsimony pressure will succeed or fail. However, the correlation between a and final performance is almost always positive, even when no parsimony is used and no trials fail. Thus, the measure a predicts more than simply which trials will fail, it also partially predicts how successfully a successful population will evolve.

Figure 4 also shows a clear, temporary drop in correlation at generation 7. This corresponds to a similar, temporary drop in the measure a seen in Figure 4. However, currently we do not have any good hypotheses to explain this fluctuation.

5 Conclusions

By examining the affects of parsimony pressure on a trial by trial basis it is evident that reports of poor average results with parsimony pressure can largely be attributed to those trials in which every individual in the population is forced to a minimal size by the parsimony function. Even with very strong parsimony pressure, populations which are not trapped in this way tend to evolve with relatively little degradation. Thus, it is clear that GP can balance performance and parsimony, but not necessarily in every trial.

It should be noted that we did not examine the generalizing ability of the programs from the successful trials. However, other research suggests that smaller programs, such as produced in the successful trials with parsimony pressure, are generally good at generalization [**?**, **?**]. Thus, it seems likely that the successful trials should also generate programs which are good at generalization.

The relationship between size and performance in an individual population (as measured by the linear coefficient a) is a useful indicator of whether that population will fail or will evolve successfully. In particular, the relationship between a and the amount of parsimony pressure α is a key factor in determining whether selection pressure is strongest for smaller individuals or for better performing individuals. Thus, this relationship is a useful tool for examining the influences of selection on an evolving population.

However, it is not simply the case that success is predicated on the relative values of a and the amount of parsimony pressure. Both the forces leading to code growth and the amount of parsimony pressure influence the value of a. Thus, there is a complex interaction between the selective forces leading to code growth, the external parsimony function, and the distribution of individuals within the evolving population. This raises the possibility of improving GP performance by manipulating these factors to improve a population's distribution, for example, by increasing diversity.

Although our focus was on understanding how parsimony pressure affects evolution rather than on improving techniques for applying parsimony pressure, our results suggest several possible improvements. First, and perhaps most simply, one could measure the value a in early generations and halt trials with a particularly low value. This would save a considerable amount of time which would otherwise be wasted on trials which are unlikely to be successful. Second, one could adjust the amount of parsimony pressure to maintain the favored relationship of $a > \alpha$, at least for as long as performance is a more important factor than size. In fact, it should be possible to adjust the amount of parsimony pressure relative to a to favor either performance or size.